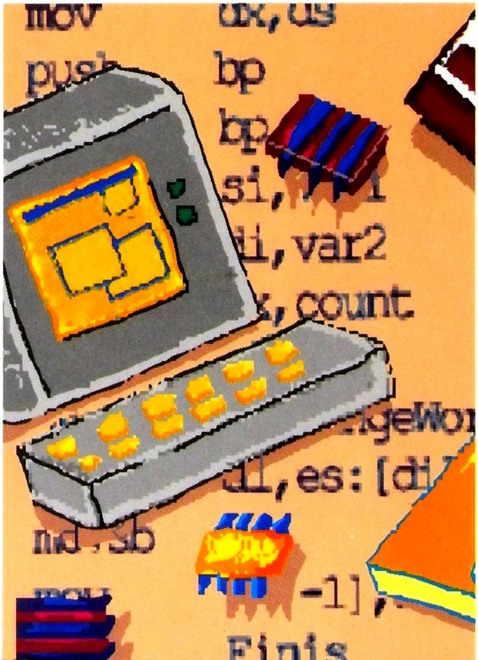
TURBO ASSEMBLER®



BORLAND

TURBO ASSEMBLER®

Borland's No-Nonsense License Statement!

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software *just like a book*, with the following single exception. Borland International authorizes you to make archival copies of the software for the sole purpose of backing-up our software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is **no possibility** of it being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time (unless, of course, Borland's copyright has been violated).

Programs that you write and compile using Turbo Assembler software may be used, given away or sold without additional license or fees, as long as all copies of such programs bear a copyright notice. By "copyright notice" we mean either your own copyright notice or, if you prefer, the statement, "Created using Turbo Assembler, Copyright © Borland 1988."

The sample programs included on the Turbo Assembler diskettes provide a demonstration of how to use the various features of Turbo Assembler. They are intended for educational purposes only. Borland International grants you (the registered owner of Turbo Assembler) the right to edit or modify these sample programs for your own use, but you may not give away or sell them, alone or as part of any program, in executable, object or source code form. You may, however, incorporate miscellaneous sample program routines into your programs, as long as your resulting programs do not substantially duplicate all or part of a sample program in appearance or functionality and all copies of such programs bear a copyright notice.

Limited Warranty

With respect to the physical diskette and physical documentation enclosed herein, Borland International, Inc. ("Borland") warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, Borland will replace the defective diskette or documentation. **If you need to return a product, call the Borland Customer Service Department to obtain a return authorization number.** The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, and special, incidental, consequential, or other similar claims.

Borland International, Inc. specifically **disclaims** all other warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the diskette and documentation, and the program license granted herein in particular, and without limiting operation of the program license with respect to any particular application, use, or purpose. **In no event shall Borland be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.**

Governing Law

This statement shall be construed, interpreted, and governed by the laws of the state of California. Use, duplication, or disclosure by the U.S. Government of the computer software and documentation in this package shall be subject to the restricted rights under DFARS 52.227-7013 applicable to commercial computer software.

1 TAUHEAD 890318

Turbo Assembler®

ユーザーズガイド

バージョン1.0

(NEC PC-9801シリーズ)
(IBM PC, XT, AT, PS/2および完全な互換機)

Copyright © 1988
All Right Reserved

Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001

編訳 (株)ボーランド ジャパン 1989

本マニュアルの一部または全部を許可なく複製または転載することはできません。

目　次

はじめに ……… 1
　必要なハードウェアとソフトウェア ……… 2
　マニュアルについて ……… 2
　　　ユーザーズガイド ……… 2
　マニュアルの表記法 ……… 4

第1章　はじめる前に ……… 5
　マスターディスクの内容 ……… 6
　Turbo Assemblerのインストール ……… 6

第2章　Turbo Assemblerのプログラミング ……… 7
　Turbo Assemblerプログラムの作成 ……… 9
　　プログラムのアセンブル ……… 9
　　プログラムのリンク ……… 11
　　プログラムの実行 ……… 12
　　復習してみよう ……… 12
　Turbo Assemblerプログラムの修正 ……… 14
　　プリンタに出力する ……… 16
　もう少し複雑なTurbo Assemblerプログラムを作る ……… 18
　　REVERSE.ASMの実行 ……… 19
　コンピュータのアーキテクチャ ……… 21
　　アセンブリ言語の特徴 ……… 22
　8088と8086プロセッサ ……… 26
　　8086の性能 ……… 26
　　メモリ ……… 27
　　入力と出力 ……… 29
　　レジスタ ……… 31
　　　フラグレジスタ ……… 32
　　　汎用レジスタ ……… 33
　　　　AXレジスタ ……… 34

BXレジスタ……35
CXレジスタ……35
DXレジスタ……37
SIレジスタ……37
DIレジスタ……38
BPレジスタ……39
SPレジスタ……40
命令ポインタ……42
セグメントレジスタ……43
CSレジスタ……47
DSレジスタ……48
ESレジスタ……48
SSレジスタ……48
8086の命令セット……49
NEC PC-9801シリーズおよびIBM PC/XT……54
入出力デバイス……54
システムソフトウェア……55
DOS……56
キーの読み込み……57
画面への文字表示……58
プログラムの終了……59
BIOS……59
ディスプレイモードの選択(IBM PCのとき)……60
場合によってはハードウェアを直接制御することも必要……61
他の資源……61

第3章　コマンドラインリファレンス……63
DOSからTurbo Assemblerを開始する……64
コマンドラインのオプション……67
/A……68
/B……68
/C……69
/D……70
/E……71
/Hまたは/?……72

/I ……………………………………………73
/J ……………………………………………74
/KH ……………………………………………75
/KS ……………………………………………76
/L ……………………………………………77
/LA ……………………………………………77
/ML ……………………………………………78
/MU ……………………………………………79
/MX ……………………………………………80
/N ……………………………………………81
/P ……………………………………………82
/R ……………………………………………83
/S ……………………………………………84
/T ……………………………………………85
/V ……………………………………………85
/W ……………………………………………86
/X ……………………………………………88
/Z ……………………………………………88
/ZD ……………………………………………89
/ZI ……………………………………………90
間接コマンドファイル……………………………………………91
コンフィギュレーションファイル……………………………………………92

第4章　アセンブラプログラミングの基本 ……………………………………………93
アセンブラプログラムの構成要素と構造……………………………………………94
行の書式……………………………………………96
ラベル……………………………………………96
命令ニーモニックと疑似命令 ……………………………………………100
END疑似命令 ……………………………………………101
オペランド ……………………………………………102
レジスタオペランド ……………………………………………103
定数オペランド ……………………………………………104
式 ……………………………………………106
ラベルオペランド ……………………………………………107
メモリアドレシングモード ……………………………………………109

コメント ……116
セグメント疑似命令 ……120
簡略セグメント疑似命令 ……120
.STACK,.CODE,.DATA ……120
DOSSEG ……124
.MODEL ……124
その他の簡略セグメント疑似命令 ……126
標準セグメント疑似命令 ……127
簡略セグメント疑似命令と標準セグメント疑似命令 ……132
データの配置 ……133
ビット，バイト，基数 ……133
10進数，2進数，8進数，16進数 ……135
デフォルト基数の選択 ……140
初期化済みデータの定義 ……142
配列の初期化 ……143
文字列の初期化 ……144
式とラベルによる初期化 ……145
初期化しないデータの定義 ……146
名前が付けられたメモリアドレス ……147
データの転送 ……151
データサイズの選択 ……152
符号付きデータと符号なしデータ ……154
データサイズの変換 ……155
セグメントレジスタのアクセス ……157
スタックのアクセス ……159
データの交換 ……159
I/O（入出力）……160
演算 ……162
算術演算 ……162
加算と減算 ……162
32ビットのオペランド ……163
インクリメントとデクリメント ……165
乗算と除算 ……166
符号の変更 ……169
論理演算 ……169

シフトとローテーション …………172
ループとジャンプ …………176
無条件ジャンプ …………176
条件ジャンプ …………179
ループ処理 …………182
サブルーチン …………186
サブルーチンの動作 …………186
パラメータの受け渡し …………190
リターン値 …………191
レジスタの退避 …………192
アセンブラのサンプルプログラム …………193

第5章　Turbo Assemblerプログラミング入門 …………199
イコール定義 …………200
EQU疑似命令 …………200
$定義済みシンボル …………205
＝疑似命令 …………206
ストリング命令 …………208
データ転送ストリング命令 …………208
LODS命令 …………208
STOS命令 …………210
MOVS命令 …………211
ストリング命令の繰り返し実行 …………212
文字列ポインタのオーバラン …………213
データスキャンストリング命令 …………214
SCAS命令 …………214
CMPS命令 …………217
ストリング命令のオペランド指定 …………218
マルチモジュールプログラム …………220
PUBLIC疑似命令 …………222
EXTRN疑似命令 …………223
GLOBAL疑似命令 …………226
インクルードファイル …………228
リストファイル …………230
アセンブル情報付きのソースコード …………231

シンボルテーブルのリスト ……235
ラベルテーブル ……235
グループとセグメントのテーブル ……236
クロスリファレンステーブル ……237
リストの内容と形式の指定 ……241
リスト行選択疑似命令 ……241
%LISTと%NOLIST ……241
%CONDSと%NOCONDS ……242
%INCLと%NOINCL ……242
%MACSと%NOMACS ……243
%CTLSと%NOCTLS ……243
リスト形式指定疑似命令 ……244
フィールド幅指定疑似命令 ……245
%PUSHLCTLと%POPLCTL ……245
他のリスト制御疑似命令 ……246
アセンブル中のメッセージ表示 ……247
ソースコードの条件アセンブル ……248
条件アセンブル疑似命令 ……249
IFとIFE ……249
IFDEFとIFNDEF ……250
他の条件アセンブル疑似命令 ……252
疑似命令のELSEIFファミリ ……254
条件エラー疑似命令 ……255
.ERR,.ERR1,.ERR2 ……255
.ERREと.ERRNZ ……256
.ERRDEFと.ERRNDEF ……256
他の条件エラー疑似命令 ……257
アセンブラプログラミングの落とし穴 ……258
DOSに戻るのを忘れる ……258
RET命令を付け忘れる ……259
リターンの型が間違っている ……260
オペランドの順番が逆になっている ……262
スタックの確保を忘れる，またはスタック領域が足りない ……263
サブルーチンコールによって必要なレジスタを破壊する ……264
条件ジャンプ命令を誤って使う ……266

ストリング命令の落とし穴 ……267
REP文字列のオーバランを忘れる ……268
CXカウンタを使えばセグメント全体をカバーできると思い込む……271
ディレクションフラグの設定が間違っている ……272
繰り返しストリング比較命令を誤って使う ……273
ストリング命令のデフォルトセグメントを忘れる ……273
バイトサイズからワードサイズへの変換に誤りがある ……275
複数のプレフィクスを使用する ……277
ストリング命令のオペランドをあてにする ……277
予期しない副作用が発生することもある ……279
乗算によりレジスタを破壊する ……279
ストリング命令がいくつかのレジスタを書きかえることもある ……280
キャリフラグを変化させない命令もある ……280
フラグの設定場所と参照場所が離れすぎている ……281
メモリオペランドと即値オペランドを混同する ……282
セグメントレジスタの一巡エラー ……284
割り込みハンドラでレジスタやフラグの退避・復帰を行なわない ……286
オペランドおよびデータテーブルにグループ
オーバライドを付け忘れる ……287

第6章　Turbo AssemblerとTurbo Cのインターフェース ……293
Turbo Cでインラインアセンブラを使う ……294
インラインアセンブラの使いかた ……296
インラインアセンブラモードの指定方法 ……301
Turbo Assemblerを起動してインラインアセンブラ
コードをアセンブルする ……302
Turbo Cはインラインアセンブラコードをどこへストアするか ……303
80186と80286の命令には-1スイッチを使う ……304
インラインアセンブラ命令文の形式 ……305
インラインアセンブラのセミコロン ……306
インラインアセンブラのコメント ……306
構造体と共用体の要素にアクセス ……307
インラインアセンブラのプログラム例 ……308
インラインアセンブラの制限 ……312
メモリオペランドとアドレスオペランドの制限 ……312

インラインアセンブラには自動変数の
デフォルトサイズ調整がない ……314
レジスタ退避の必要性 ……315
関数の呼び出しとレジスタ変数の退避 ……315
内部レジスタ変数の消滅 ……315
インラインアセンブラと純粋なCの欠点 ……316
移植性(可搬性)と保守性が悪くなる ……316
コンパイル速度の低下 ……316
TCCのみで使用可能 ……317
最適化が犠牲となる ……317
エラートレースの制限 ……317
デバッグの制限 ……318
最初はCだけを使い, 最終版にインラインアセンブラも使う ……318
Turbo CからTurbo Assembler関数を呼び出す ……320
Turbo CとTurbo Assemblerのインターフェースの概要 ……322
メモリモデルとセグメント ……322
簡略セグメント疑似命令とTurbo C ……323
従来のセグメント疑似命令とTurbo C ……324
セグメントデフォルト：セグメントはいつセットすべきか ……327
パブリック変数と外部変数 ……330
下線記号 ……330
大文字と小文字の区別 ……331
ラベルの型 ……332
far外部変数はセグメントの外に置く ……333
リンカのコマンドライン ……335
Turbo AssemblerとTurbo Cの相互作用 ……335
パラメータの受け渡し ……335
レジスタの退避 ……343
値の返し方 ……343
Turbo CからTurbo Assembler関数を呼び出す ……345
Pascalの呼び出し慣例 ……348
Turbo AssemblerからTurbo Cを呼び出す ……350
Cスタートアップコードのリンク ……350
セグメントの設定が正しいことを確認する ……351
呼び出しの実行 ……351

Turbo AssemblerモジュールからTurbo C関数を呼び出す……352

第7章　Turbo AssemblerとTurbo Pascalのインターフェース 355

Turbo Pascalのメモリマップ……356
プログラムセグメントプレフィクス……358
コードセグメント……358
グローバルデータセグメント……358
スタック……359
ヒープ……359
Turbo Pascalのレジスタの使い方……361
nearかfarか……362
Pascalのプログラムとアセンブラルーチンの情報の共用……363
$Lコンパイラ指令とexternalサブプログラム……363
PUBLIC疑似命令：アセンブラルーチンの情報を
Pascalプログラムが利用できるようにする……364
EXTRN疑似命令：Pascalプログラムの情報を
アセンブラルーチンが利用できるようにする……364
EXTRN宣言を使用するときの制約……367
セグメント参照の解決の利用……368
デッドコードの削除……368
Turbo Pascalのパラメータの引き渡し方法……370
値パラメータ……370
スカラ型……370
実数……371
Single, Double, Extended, Comp 8087型……371
ポインタ……371
文字列……371
レコードと配列……372
集合……372
変数パラメータ……372
スタックの管理……372
パラメータへのアクセス……373
BPレジスタを使ってスタック内のアドレスを指定する方法……374
ARG疑似命令……374
.MOEDLとTurbo Pascal……376

他のレジスタを使ってスタック内のアドレスを指定する方法 …………376
Turbo Pascalの関数の結果…………378
スカラ型の結果 …………378
実数型の結果 …………378
8087型の結果 …………378
文字列型の結果 …………378
ポインタ型の結果 …………378
ローカル変数の格納領域の確保 …………379
静的格納領域の確保 …………379
揮発格納領域の確保 …………380
Turbo Pascalのプログラムとリンクするアセンブリ言語ルーチンの例…………382
汎用の16進数への変換ルーチン …………382
2つの変数の交換…………385
DOSの環境変数の検索 …………388

第8章　Turbo AssemblerとTurbo Basicのインターフェース…393
パラメータの受け渡し …………395
カレントデータセグメント外の変数 …………397
どのCALLを使うべきか …………398
スタックのポップ …………399
アセンブラプログラムの作成 …………399
インラインアセンブラプロシージャの呼び出し …………400
メモリの中のTurbo Basicのルーチンを探す …………403
文字列を隠す …………404
CALL ABSOLUTE …………405
固定アドレスに対するCALL ABSOLUTE …………406
変動アドレスに対するCALL ABSOLUTE …………406
CALL ABSOLUTEのその他の問題点 …………408
CALL INTERRUPT …………409
例題プログラム …………411

第9章　Turbo AssemblerとTurbo Prologのインターフェース 413
外部述語の宣言 …………414
呼び出し規則とパラメータ …………415
名前 …………415

アセンブリ言語の述語を書く ……417
double述語の実現 ……419
複数のフローパターンを持つ述語の実現 ……423
アセンブリ関数からTurbo Prologの述語を呼び出す ……425
リストとファンクタ ……427

第10章　Turbo Assemblerの応用プログラミング ……431
セグメントオーバライドプレフィクス ……432
セグメントオーバライドプレフィクスの別の形式 ……434
セグメントオーバライドプレフィクスが使えない場合 ……435
複数セグメントにアクセスする ……436
ローカルラベル ……437
自動ジャンプサイズ決定 ……441
コードとデータへの前方参照 ……446
繰り返しブロックとマクロ命令の使用 ……450
繰り返しブロック ……450
繰り返しブロックと変数パラメータ ……453
マクロ命令 ……454
マクロ命令のネスト ……459
マクロ命令と条件疑似命令 ……460
EXITMで展開を停止する ……461
マクロ命令内でラベルを定義する ……462
任意のデータ構造を扱う ……465
STRUC疑似命令 ……465
STRUCを使うことのメリットとデメリット ……469
固有の構造体フィールド名 ……470
構造体のネスト ……470
構造体の初期化 ……471
RECORD疑似命令 ……472
レコードのアクセス ……475
WIDTH演算子 ……476
MASK演算子 ……477
レコードの用途 ……478
UNION疑似命令 ……480
セグメント疑似命令 ……483

SEGMENT疑似命令……483
セグメント名とアラインメントフィールド……483
combineフィールド……484
useフィールドとclassフィールド……486
セグメントのサイズ，型，名前，ネスティング……486
セグメントの順序づけ……487
GROUP疑似命令……489
ASSUME疑似命令……491
簡略セグメント疑似命令……495
マルチセグメントプログラムのサンプル……500

第11章　80386と他のプロセッサ……505
アセンブラコード内でプロセッサの型を切り換える……506
80186と80188……507
80186命令のアセンブルを有効にする……507
新しい命令……507
PUSHAとPOPA……507
ENTERとLEAVE……508
BOUND……509
INSとOUTS……510
8086命令の拡張版……511
即値の直接プッシュ……512
即値のシフトとローテート……512
即値の乗算……513
80286……515
80286アセンブルモードにする……516
80386……517
80386アセンブルモードの選択……517
80386のセグメント型……518
簡略セグメント疑似命令と80386のセグメント型……522
48ビットのFWORDデータ型……523
80386のレジスタ構成……526
32ビット汎用レジスタ……527
32ビットフラグレジスタ……528
32ビット命令ポインタ……528

80386のセグメントレジスタ……529
80386のアドレシングモード……531
80386の命令……536
ビットのテスト……536
ビットのスキャン……537
符号付き/符号なし拡張を伴うデータ転送……538
DWORDまたはQWORDへのデータ変換……539
複数ワードにまたがるシフト……540
バイトの条件セット……542
SS, FS, GSの設定……543
拡張命令……544
特殊形式のMOV命令……544
8086命令の32ビットバージョン……545
LOOPとJCXZの32ビットバージョン……546
ストリング命令の32ビットバージョン……547
IRETD……548
PUSHFDとPOPFD……549
PUSHADとPOPAD……549
IMUL命令の32ビットバージョン……549
16ビットと32ビットの命令とセグメントの混在……550
80386の関数例……553
80287……557
80387……558

第12章　Turbo AssemblerのIdealモード……559
Idealモードとは何か？……560
なぜIdealモードを使うのか？……561
Idealモードの開始と終了……562
MASMモードとIdealモードの違い……564
Idealモードのトークン……564
シンボルトークン……564
重複メンバ名……565
浮動小数点トークン……566
文字列と数値のイコール定義(EQU疑似命令と=疑似命令)……566
式とオペランド……567

大カッコ演算子 ……567
オペランドの例 ……568
演算子 ……570
構造体メンバ内のピリオド ……570
構造体へのポインタ ……570
SYMTYPE演算子 ……571
HIGH演算子とLOW演算子 ……571
任意指定のPTR演算子 ……572
SIZE演算子 ……573
疑似命令 ……574
リスト制御疑似命令 ……574
ピリオド(.)で始まる疑似命令 ……574
順序が反転した疑似命令とシンボル名 ……575
引用符で囲んだ文字列を伴う疑似命令 ……576
セグメントとグループ ……577
グループに属するセグメント内データのアクセス ……577
near型またはfar型のコードラベルの定義 ……579
外部シンボル,パブリックシンボル,グローバルシンボル ……580
その他の相違点 ……581
余計な領域を確保しない ……581
BOUND命令のオペランド ……582
マクロ内のコメント ……582
ローカルシンボル ……583
MASMモードとIdealモードのプログラミングの比較 ……584
MASMモードのサンプルプログラム ……584
Idealモードのサンプルプログラム ……586
MASMモードとIdealモードの分析 ……587

参考文献 ……591

索引 ……595

はじめに

ボーランド社のTurbo Assemblerへようこそ。Turbo Assemblerは前方参照の解決を採用した1パス方式のアセンブラで，最高48,000行/分（IBM PS/2のモデル60）のアセンブル速度，MASMとの互換機能，Idealモードの拡張シンタックス（オプション）などの優れた特長を備えています。アセンブリ言語のプログラム開発を容易に行なうための数々の便利機能は，入門者にも上級プログラマにも役に立つものと確信しております。以下にTurbo Assemblerの特長を簡単に紹介します。

- ■80386をフルサポート
- ■文法の型チェック機能
- ■簡略セグメント疑似命令
- ■リスト機能
- ■ローカルラベル
- ■ローカルスタックシンボルと引数呼び出し機能
- ■構造体と共用体
- ■ネスト可能な疑似命令
- ■MASMをエミュレートする*Quirks*モード
- ■Turbo Debugger用のソース出力
- ■組み込みクロスリファレンスユーティリティ（TCREF）
- ■コンフィギュレーションファイルとコマンドファイル

Turbo Assemblerはソースファイル（.ASM）からオブジェクトファイル（.OBJ）モジュールを生成するパワフルなコマンドラインアセンブラです。ボーランド社の高速リンカを使えば複数のオブジェクトモジュールをリンクして実行可能ファイル（.EXE）を作ることができます。

Turbo Assemblerは80x86(8086, 80186, 80286, 80386)および80x87(8087, 80287, 80387)のプロセッサファミリ用に設計されています（80x86および80x87のプロセッサファミリの命令セットの詳細については，インテル社のデータブックを参照してください）。

必要なハードウェアとソフトウェア

Turbo AssemblerはNEC PC-9801シリーズとIBMのPCファミリ（XT, AT, PS/2, およびそれらの互換機）で使うことができます。OSはMS-DOS（バージョン2.0以上），必要なメモリ容量は256Kバイトです。

Turbo Assemblerがサポートするプロセッサは8086, 80186, 80286, および80386です。数値演算コプロセッサ8087, 80287, 80387用の浮動小数点命令もサポートしています。

マニュアルについて

Turbo Assemblerには「**ユーザーズガイド**」（このマニュアル）と「**リファレンスガイド**」の2冊のマニュアルが添付されています。ユーザーズガイドはTurbo Assemblerの操作の基本とアセンブリ言語のプログラミングについての詳細を解説しています。**リファレンスガイド**は演算子，定義済みシンボル，およびTurbo Assemblerの疑似命令について解説しています。

以下に**ユーザーズガイド**の詳細を説明します。

ユーザーズガイド

第1章　はじめる前に

マスターディスクのファイルとTurbo Assemblerのユーザのシステムへのインストール方法を説明します。

第2章　Turbo Assemblerのプログラミング

アセンブリ言語のプログラミングの概要を説明します。またサンプルプログラムを使って便利なコマンドラインのスイッチについて紹介します。コンピュータの基礎や8088プロセッサに関して説明します。

第3章　コマンドラインリファレンス

すべてのコマンドラインのオプション，コンフィギュレーションファイルとコマンドファイルの使用法を説明します。

第4章　アセンブラプログラミングの基礎

疑似命令，機械命令，メモリのアクセス，セグメントなどのアセンブラの基本要素を解説します。

第5章　Turbo Assemblerプログラミング入門

Turbo Assemblerの疑似命令，文字列命令など，第4章の内容をさらに詳しく説明します。アセンブリ言語のプログラミングの共通の落し穴についても説明します。

第6章　Turbo AssemblerとTurbo Cのインターフェース

高級言語のTurbo Cとアセンブリ言語を組み合わせて使う方法を解説します。またアセンブラモジュールとTurbo Cのリンク方法やTurbo CからTurbo Assemblerの関数を呼び出す方法を解説します。

第7章　Turbo AssemblerとTurbo Pascalのインターフェース

Turbo Pascalとアセンブリ言語を組み合わせて使う方法をサンプルプログラムを使って解説します。

第8章　Turbo AssemblerとTurbo Basicのインターフェース

Turbo Basicとアセンブリ言語を組み合わせて使う方法をサンプルプログラムを使って解説します。

第9章　Turbo AssemblerとTurbo Prologのインターフェース

Turbo Prologとアセンブリ言語を組み合わせて使う方法をサンプルプログラムを使って解説します。

第10章　Turbo Assemblerの応用プログラミング

セグメントオーバーライドプレフィクス，マクロ命令，セグメント疑似命令など，1章～9章の内容をさらに詳しく解説します。

第11章　80386と他のプロセッサ

80386のプログラミングについて説明します。

第12章　Turbo AssemblerのIdealモード

Idealモードとその使用法を解説します。

参考文献

アセンブリ言語のプログラミングについて役に立つ書籍を紹介します。

マニュアルの表記法

NEC PC-9801, IBM PC, またはそれらの互換機について言及したときは, 8088, 80186, 80286, 80386（これらはすべて80x86と共通表記します）を使ったコンピュータを示します。MS-DOS（IBM機以外の機種で採用されているオペレーティングシステムのタイプ), PC-DOS (IBM機に採用されているオペレーティングシステムのタイプ) などはすべてバージョン2.0以上のOSを示しています。

このマニュアルでは次のように書体を使いわけています。

Italics　ラベル, 変数, 配列を示します。文法説明ではユーザ指定項目はイタリック体で表記しています。

Boldface　疑似命令, 機械命令, シンボル, 演算子, コマンドラインオプションはボールド体で表記しています。

CAPITALS　機械命令, 疑似命令, レジスタ, 演算子はアルファベットの大文字で表記しています。

KEY　キーボード上のキー（***ESC, F10***など）はこの書体で表記しています。

第1章

はじめる前に

アセンブラのプログラミングをはじめる前に1つだけやっておかなければならないことがあります。DOS上でTurbo Assemblerの各提供ディスクのバックアップをとって「作業用ディスク」を作ってください。オリジナルディスクは大切に保管してください（ディスクを交換する場合は有料となります。オリジナルディスクはバックアップをとって作業用ディスクを作るときだけ使ってください）。

MASMからTurbo Assemblerへ移行する方は付録Bを参照してください。Turbo AssemblerとMASMの相違点がわかります。

マスターディスクの内容

提供ディスクの内容は次のとおりです。

■TASM.EXE（Turbo Assembler）
■TLINK.EXE（Turbo Linker）
■MAKE.EXE（コマンドラインのMAKEユーティリティ）
■TLIB.EXE（ライブラリ）
■README.COM（READMEファイルを表示するプログラム）
■README（ソフトウェアとドキュメントに関する詳細情報）
■TCREF.EXE（ソースファイルのクロスリファレンスユーティリティ）
■OBJXREF.COM（オブジェクトファイルのクロスリファレンスユーティリティ）
■GREP.COM（ファイル検索ユーティリティ）
■TOUCH.COM（ファイルのアップデートユーティリティ）

Turbo Assemblerのインストール

Turbo Assemblerのインストールは簡単です。ハードディスクをお持ちの場合はTurbo Assembler用のディレクトリを作ってそこへTASM.EXEをコピーしてください。フロッピィディスクのみ使う場合はフロッピィディスクへTASM.EXEをコピーしてください。

次に使いたいユーティリティを同じディレクトリにコピーすればインストールは完了です。次章ではTASMを使ったプログラミングの基本を説明します。

第2章

Turbo Assemblerのプログラミング

もし今までにアセンブリ言語プログラミングの経験がなければ，今がそれをはじめるいい機会です。アセンブリ言語はハッカーやマニアだけが使うプログラミング言語だという人もいますが，そんな考えは今すぐ捨ててください。非常にロジカルなコンピュータ言語を人間にわかりやすい形にしたものがアセンブリ言語なのです。アセンブリ言語は非常にパワフルな言語です。事実，NEC PC-9801シリーズやIBM PCファミリ，またはその互換機の心臓部にあるプロセッサ80x86ファミリの性能をフルに引き出せるのはアセンブリ言語だけです。

プログラムを作る場合，基本的にはアセンブリ言語だけでこと足ります。必要ならアセンブリ言語をTurbo C, Turbo Pascal, Turbo Prolog, Turbo Basicなどの他の高級言語プログラム内に組み入れることもできます。いずれにしろ，アセンブリ言語を使えばコンパクトで高速のプログラムを作ることができます。処理速度と同じくらい重要なことですが，アセンブリ言語はシステムクロックなどコンピュータに関するあらゆるレベルの動作をコントロールすることができます。

この章ではアセンブリ言語の概要を紹介し，アセンブリ言語のプログラミングの特徴を説明します。はじめにアセンブリ言語のプログラムを入力・実行して，アセンブリ言語の感触を味わうとともにアセンブラの使い方を学びます。次にコンピュータの基本，特に8086プロセッサの基本について説明します。8086に対してアセンブリ言語がいかに威力を発揮するかを理解することができます。最後にNEC PC-9801やIBM PCなどに関係したアセンブリ言語プログラミングについて説明します。

第4章「アセンブラプログラミングの基本」ではアセンブリ言語プログラムの構造，プログラムの基本要素などこの章で省いたことがらを説明し，豊富なプログラム例を通してこの章と第4章で学んだことがらをまとめます。

第5章「Turbo Assemblerプログラミング入門」ではアセンブリ言語プログラミングの基礎を説明します。第10章「Turbo Assemblerの応用プログラミング」ではメモリモデル，マクロ命令，および他の応用知識を学習します。

このマニュアルの一部を読んだだけですぐにアセンブリ言語の上級プログラマになるこ

とはできません。このマニュアルはアセンブリ言語の入門書でありプログラム作成の手引書として書かれています。アセンブリ言語のプログラミングとNEC PC-9801やIBM PCのアーキテクチャについては，多くの著書が出版されていますのでそちらを読むことをお勧めします（このマニュアルの巻末の参考文献を参照してください。）。

これらのマニュアルはIBMパーソナルコンピュータのシステムソフトウェアとハードウェアとアセンブリ言語のプログラミングインタフェースについて解説しています。

先へ進む前に，第3章「コマンドラインリファレンス」を読んでコマンドラインのオプションに慣れておくとよいでしょう。もしTurbo Assemblerのインストールが終ってなければ，第1章「はじめる前に」で説明したように，Turbo Assemblerをインストールしておきましょう（Turbo Assemblerのオリジナルディスクから作業用ディスクを作るか，あるいはファイルをハードディスク上にコピーします）。

アセンブリ言語は複雑な体系をもっているため，たとえ簡単なプログラムを作る場合でも多くの知識が要求されます。サンプルプログラムの中には説明していない機能を使うこともありますが，これは**やむをえない**ことです。詳細は順を追って説明します。特定の機能について詳しく知りたい場合は**リファレンスガイド**の第3章の「疑似命令」の節を参照してください。

ではアセンブリ言語のプログラムを作りましょう。リファレンスガイドを手元に置いて，第3章をいつでも開けるようにしておいてください。

Turbo Assemblerプログラムの作成

プログラミングの世界では Hello, world というメッセージを表示するプログラムが入門者が最初に作るサンプルと相場は決っています。ここでもその慣例に従うことにしましょう。ASCIIコード用のテキストエディタを使って下のサンプルプログラムを入力してHELLO.ASMというファイルにセーブしてください。

```
    DOSSEG
    .MODEL  SMALL
    .STACK  100h
    .DATA
 HelloMessage DB 'Hello, world',13,10,'$'
    .CODE
    mov  ac,@data
    mov  ds,ax                          ; DSをデータセグメントのポイントにセット
    mov  ah,9                           ; 文字列プリントのDOSファンクション番号
    mov  dx,OFFSET HelloMessage         ; "Hello world"にポイント
    int  21h                            ; "Hello world"を表示
    mov  ah,4ch                         ; プログラム終了のDOSファンクション番号
    int  21h                            ; プログラム終了
    END
```

C言語またはPascalのプログラミング経験がある方は，上のサンプルプログラムの大きさに驚くかもしれません。アセンブリ言語の各命令が実行する処理内容は，CやPascalのそれと比べて一般的に少ないのです。しかし，これらアセンブリ言語の命令はどのように組み合わせて使おうがあなたの自由です。言い換えるとアセンブリ言語を使えばCやPascalではできないこともコンピュータにやらせることができるということです。

プログラムのアセンブル

HELLO.ASMをセーブしたらさっそく実行してみましょう。プログラムを実行するためにはHELLO.ASMを実行可能な形式に変換してやらなくてはなりません。このためには図2.1に示すようにアセンブルとリンクという2つのステップが必要となります。この図は，編集，アセンブル，リンク，実行というプログラム開発のすべての手順を表わしています。

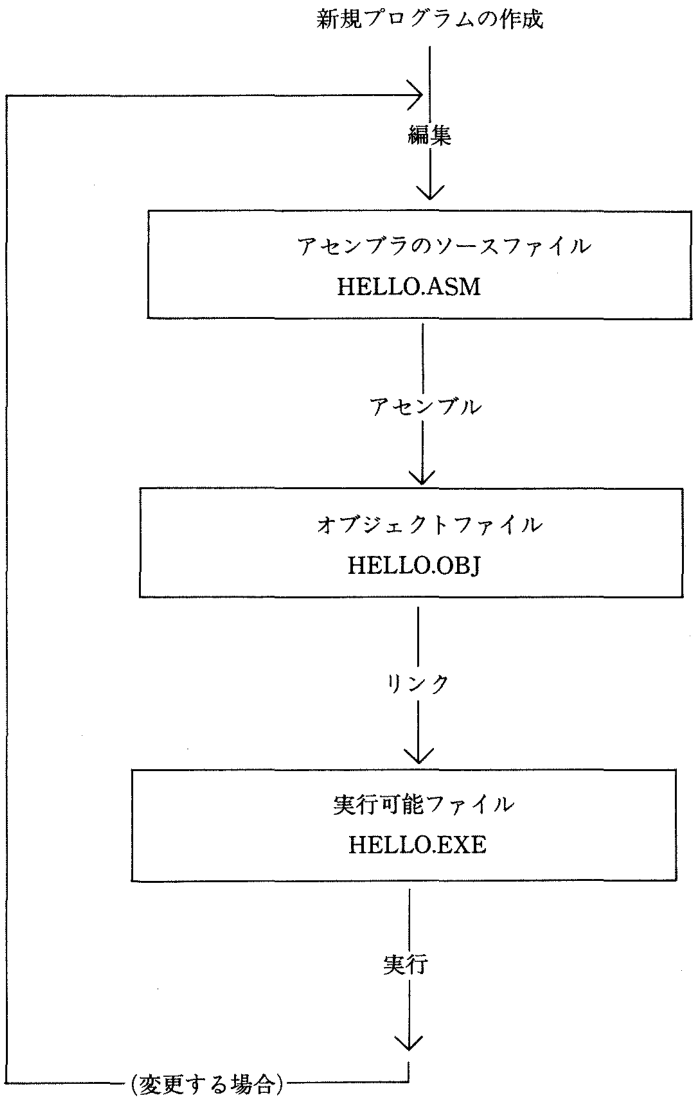


図2.1　編集，アセンブル，リンク，実行のサイクル

アセンブルのステップではソースコードをオブジェクトモジュールという中間コードに変換します。リンクのステップでは複数の**オブジェクトモジュール**を結合して1つの実行可能プログラムを作ります。これらアセンブルとリンクはコマンドラインから実行すること

ができます。

HELLO.ASMをアセンブルするには次のように入力します。

```
TASM hello
```

特にファイル名について指定しない限りHELLO.ASMのオブジェクトファイル名はHELLO.OBJになります(ファイル名の拡張子は省略してかまいません。省略時はTurbo Assemblerは拡張子ASMを持つファイルをアセンブルします)。次に画面のようすを示します。

```
Turbo Assembler Version 1.0 Copyright (C) 1988 by Borland International, Inc.

Assembling file: HELLO.ASM
Error messages: None
Warning messages: None
Remaining memory: 266K
```

上の例のようにHELLO.ASMと指定しても何の問題も起こりません。問題が発生したときは警告メッセージやエラーが発生箇所を示す行番号とともに画面に表示されます。この場合はコードが上記と同じかどうかをチェックして再度プログラムをアセンブルし直してください。

プログラムのリンク

HELLO.ASMのアセンブルが無事終了したら実行まではもう1ステップだけです。いまアセンブルしたオブジェクトコードをリンクして実行可能コードを作ればよいのです。

プログラムをリンクするにはTurbo Assemblerの付属リンカのTLINKを使います。コマンドラインで次のように入力します。

```
TLINK hello
```

ここでも拡張子は入力する必要はありません。TLINKがOBJという拡張子を付けてくれます。リンクが終了すると(ほんの数秒しか，かかりません)EXEファイルが作られます。特に指定のない限り，このEXEファイルにはオブジェクトファイルと同じファイル名が付けられます。リンクが無事終了すると次のメッセージが画面に表示されます。

```
Turbo Linker Version 2.0 Copyright (c) 1987, 1988 by Borland International, Inc.
```

このサンプルプログラムでは正常にリンクが終了しますが，プログラムによってはリンク中にエラーが起ることもあります。リンクエラーが発生したときは画面に表示されます。この場合はコードを訂正し，もう一度アセンブルとリンクをやり直してください。

プログラムの実行

ではプログラムを実行しましょう。DOSのプロンプトでhelloと入力してください。次のメッセージが表示されます。

```
Hello, world
```

以上で作業は終わりです。Turbo Assemblerのプログラム作成から実行までをひととおり終了しました。

復習してみよう

HELLO.ASMの入力と実行を経験したところで，テキストの入力からプログラムの実行までのステップをもう一度復習してみましょう。

テキストエディタを使ってソースコードを入力した時点ではテキストはメモリ上にあります。この状態ではコンピュータの電源が何かの原因で切れると，ソースコードはすべて消えてしまいます。そんな悲惨なことにならないように，頻繁にソースコードをディスクにセーブすることをお勧めします。ソースコードをディスクにセーブすればファイルHELLO.ASMにテキストがコピーされます。こうすればもうコンピュータの電源を切っても大丈夫です（しかしディスクそのものが破壊された場合はファイルHELLO.ASMを復旧する手立てはありません。ディスクは定期的にバックアップをとっておきましょう）。HELLO.ASMは標準ASCIIテキストファイルです。DOSプロンプトで次のコマンドを入力すればその内容をみることができます。

```
type hello.asm
```

テキストエディタを使えば内容を編集することができます。

HELLO.ASMをアセンブルするとHELLO.ASMのテキスト命令がバイナリの命令に変換されHELLO.OBJにセーブされます。HELLO.OBJはソースファイルと実行ファイルの中間ファイルです。HELLO.OBJには，HELLO.ASMの命令から実行可能ファイルを作るために必要なすべての情報が入っています。HELLO.OBJを他のオブジェクトファイルと結合して1つのプログラムを作ることができます。第5章「Turbo Assemblerプログラミング入門」では，大きなプログラムを作る際に複数のオブジェクトファイルを結合することがいかに便利かを説明します。

TLINKを使ってHELLO.OBJをリンクすると，オブジェクトファイルHELLO.OBJは実行可能ファイルHELLO.EXEに変換されます。プロンプトで**hello**と入力すればHELLO.EXEを実行することができます。では次のようにコマンドを入力してください。

```
dir hello.*
```

ディスク上のHELLOファイルの一覧が表示されます。HELLOファイルはHELLO.ASM, HELLO.OBJ, HELLO.EXE, HELLO.MAPです。

Turbo Assemblerプログラムの修正

ではエディタへ戻ってください。今度はキーボードからデータを読み込むようにプログラムを修正してみましょう。プログラムを次のように修正してください。

```
    DOSSEG
    .MODEL  SMALL
    .STACK  100h
    .DATA
 TimePrompt  DB 'Is it after 12 noon (Y/N)?$'
 GoodMorningMessage LABEL BYTE
    DB 13,10,'Good morning, world!',13,10,'$'
 GoodAfternoonMessage LABEL BYTE
    DB 13,10,'Good afternoon, world!',13,10,'$'
    .CODE
    mov  ax,@data
    mov  ds,ac                           ; DSにデータセグメントをセット
    mov  dx,OFFSET TimePrompt            ; タイムプロンプトのアドレスをロード
    mov  ah,9                            ; 文字列プリントのDOSファンクション番号
    int  21h                             ; タイムプロンプトを表示
    mov  ah,1                            ; 文字読み込みのDOSファンクション番号
    int  21h                             ; 1文字読み込み
    cmp  al,'y'                          ; タイプした文字は小文字のy?
    jz   IsAfternoon                     ; yes (午後)
    cmp  al,'y'                          ; タイプした文字は大文字のy?
    jnz  IsMorning                       ; no (午前)
 IsAfternoon:
    mov  dx,OFFSET GoodAfternoonMessage  ; 午後のあいさつのアドレスをロード
    jmp  DisplayGreeting
 IsMorning:
    mov  dx,OFFSET GoodMorningMessage    ; 午前のあいさつのアドレスをロード
 DisplayGreeting:
    mov  ah,9                            ; 文字列プリントのDOSファンクション番号
    int 21h                              ; 適切なあいさつを表示
    mov  ah,4ch                          ; プログラム終了のDOSファンクション番号
    int  21h                             ; プログラムを終了
    END
```

このプログラムには2つの新しい重要な要素が加わっています。それは入力と判断です。このプログラムははじめに**午後か**どうか聞いてきます。そして1文字の応答をキーボードから読み込みます。文字が小文字または大文字のYであれば午後に適した挨拶を表示し，そうでなければ午前中の挨拶を表示します。このプログラムにはプログラミングの基本要素である文字の読み込み，データ処理，判断，文字の表示などが含まれています。

変更したプログラムをディスクにセーブしてください(元のHELLO.ASMは変更したプログラムに置き換わります)。次に前の例で行なったようにアセンブルとリンクを行ないます。そしてDOSプロンプトで**hello**と入力してプログラムを実行してください。次のメッセ

ージが表示されます。

```
Is it after 12 noon (Y/N)?
```

カーソルは　の後で点滅しています。キーボードからの応答を待っています。**Y**を押すと次のようなメッセージが表示されます。

```
Good afternoon, world!
```

HELLO.ASMは会話形式で判断を行なうプログラムになりました。

アセンブラプログラミングの学習過程では，キー入力のミスや文法エラーなどいろいろな間違いが見つかります。Turbo Assemblerはプログラムのアセンブル時にミスを見つけるとメッセージを表示します。メッセージには警告メッセージとエラーメッセージの2種類があります。**警告**メッセージはプログラム中に疑わしいが致命的ではないものがあったときに表示されます。このメッセージは無視できることもありますが，やはり原因をチェックしておく方がよいでしょう。**エラー**メッセージは明らかな間違いが見つかってアセンブルの続行とオブジェクトコードの生成が不可能なときに表示されます。

言い換えると，警告メッセージは注意を与えるだけで致命的ではありませんが，エラーメッセージが表示された場合はその原因を取り除かないとプログラムを実行することはできません。Turbo Assemblerの警告メッセージとエラーメッセージについては**リファレンスガイド**の付録Eに載っています。

他のプログラミング言語と同様に，Turbo Assemblerも論理エラーを見つけることはできません。Turbo Assemblerはプログラムがアセンブル可能かどうかを判断するだけで，そのプログラムがプログラマの思いどおりに動くかどうかまでは判断することができません。その判断ができるのはプログラマだけです。

サンプルプログラムがいまうまく動かなくても心配する必要はありません。他の言語で経験をつんだプログラマでさえも，8086のアセンブリ言語に慣れるのには時間がかかります。8086のアセンブリ言語ほどやっかいなものはありません。ここまでくればアセンブラプログラムがどんなものか感じがつかめてきたことでしょう。この章の残りと第4章「アセンブラプログラミングの基本」でサンプルプログラムの各要素を解説します。

プログラムのプリント方法についてはテキストエディタのマニュアルやMS-DOSユーザーズリファレンスマニュアルを参照してください。Turbo Assemblerのソースファイルは標準ASCIIテキストファイルです。したがってDOSプロンプトからPRINTコマンドを使ってソースファイルをプリントすることができます。

プリンタに出力する

プリンタは最も身近な出力デバイスです。プログラムファイルをプリンタへ出力するだけでなく，プログラムでプリンタに印字させたいことがしばしばあります。次のプログラムは，データを画面ではなくプリンタへ出力する Hello, world プログラムです。

```
    DOSSEG
    .MODEL   SMALL
    .STACK   100h
    .DATA
HelloMessage DB 'Hello, world',13,10,12
HELLO_MESSAGE_LENGTH EQU $ - HelloMessage
    .CODE
    mov   ax,@data
    mov   ds,ax                          ; DSにデータセグメントをセット
    mov   ah,40h                         ; デバイス書込みのDOSファンクション番号
    mov   bx,4                           ; プリンタを指定
    mov   cx,HELLO_MESSAGE_LENGTH        ; プリント文字数
    mov   dx,OFFSET HelloMessage         ; プリント文字列
    int   21h                            ; "Hello, world"をプリント
    mov   ah,4ch                         ; プログラム終了のDOSファンクション番号
    int   21h                            ; プログラム終了
    END
```

この Hello, world プログラムでは画面に文字列を表示するDOS関数が任意のデバイス（この場合はプリンタ）に文字列を送るDOS関数に置き換わっています。プログラムを上のように変更して実行してみましょう。プリンタに Hello, world とプリントされるのがわかります(実行する前に必らずプログラムをセーブしておいてください。ファイルHELLO.ASMは書き換えられ前の内容はなくなります)。(PC-9801シリーズのMS-DOS3.3以降の製品ではデバイスドライバPRINT.SYSをCONFIG.SYSに設定してから実行してください)。

プログラムを元に戻してプリンタではなく画面に文字列を出力する場合は，

```
mov   bx,4        ; プリンタを指定
```

を

```
mov   bx,1        ; 標準出力を指定
```

に変更します。

変更したら再びアセンブルとリンクを行なって実行してください。画面に表示される文字列の最後の文字が汎用記号 ♀ になっています（IBM PCのとき）。これは Hello, world

をプリントしたあと用紙を送るためにプログラムがプリンタに送った**改ページコード**です。このコードを画面に送ったためにコンピュータの文字セットの対応文字が表示されたのです(PC-9801シリーズでは改ページコード0chを画面に送り込むとカーソルを右に移動する動作を行なうので画面には何も影響がないように見えます)。

もう少し複雑なTurbo Assemblerプログラムを作る

次にもう少し複雑な処理を行なうプログラムを入力して実行してみましょう。テキストエディタを起動して次のプログラムを入力し，ファイルREVERSE.ASMにセーブしてください。

```
   DOSSEG
   .MODEL  SMALL
   .STACK  100h
   .DATA
MAXIMUM_STRING_LENGTH  EQU  1000
StringToReverse   DB  MAXIMUM_STRING_LENGTH DUP(?)
ReverseString     DB  MAXIMUM_STRING_LENGTH DUP(?)
   .CODE
   mov  ax,@data
   mov  ds,ax                         ; DSにデータセグメントをセット
   mov  ah,3fh                        ; ハンドラからのリードのDOSファンクション番号
   mov  bx,0                          ; 標準入力を指定
   mov  cx,MAXIMUM_STRING_LENGTH      ; 最大文字数まで読み込み
   mov  dx,OFFSET StringToReverse     ; プリント文字列をストア
   int  21h                           ; 文字列読み込み
   and  ax,ax                         ; 文字は読み込まれたか？
   jz   Done                          ; no（読み込み終了）
   mov  cx,ax                         ; cxに文字列の長さをセット(カウンタ)
   push cx                            ; 文字列の長さをセーブ
   mov  bx,OFFSET StringToReverse
   mov  si,OFFSET ReverseString
   add  si,cx
   dec  si                            ; 逆順文字列バッファの終わりをポイント

ReverseLoop:
   mov  al,[bx]                       ; 次の文字読み込み
   mov  [si],al                       ; 逆順に文字をストア
   inc  bx                            ; 次の文字をポイント
   dec  si                            ; 直前の場所をポイント
   buffer
   loop ReverseLoop                   ; 次の文字を移動
   pop  cx                            ; 文字列の長さを復帰
   mov  ah,40h                        ; ハンドラからの書込みDOSファンクション番号
   mov  bx,1                          ; 標準出力を指定
   mov  dx,OFFSET ReverseString       ; 文字列をプリント
   int  21h                           ; 逆順文字列をプリント
Done:
   mov  ah,4ch                        ; プログラム終了のDOSファンクション番号
   int  21h                           ; プログラム終了
   END
```

このプログラムが何をするものかもうおわかりでしょう。ではいつものようにプログラムをディスクへセーブしてください。

REVERSE.ASMの実行

REVERSE.ASMを実行するにはアセンブルしなければなりません。次の順番でコマンドを入力してください。実行可能ファイルが作られます。

```
TASM reverse
TLINK reverse
```

DOSプロンプトで**reverse**と入力すればプログラムを実行できます。Turbo Assemblerが何らかのエラーメッセージや警告メッセージを表示した場合は，プログラムが上記のリストどおりに入力されているかチェックしてからもう一度実行してください。

プログラム実行後はカーソルが画面上で点滅しています。キー入力モードになっていますので次のように入力してください（最後に**リターンキー**を押します)。

```
ABCDEFG
```

次のような文字列が表示されます。

```
GFEDCBA
```

コマンドラインで再び**reverse**と入力し，次のように入力してください(最後に**リターンキー**を押します)。

```
0123456789
```

次のように文字列が表示されます。

```
9876543210
```

これでREVERSE.ASMの機能がおわかりでしょう。これはキーボードから入力した文字列を逆順に並びかえて表示するプログラムです。後の章でも説明しますが，文字や文字列の高速処理はアセンブリ言語の優れた特長の1つなのです。

サンプルプログラムを使ってプログラムの入力，アセンブル，リンク，実行を実習してきましたが，これでアセンブラプログラミングの基本である入力，処理，出力を体験したことになります。

オブジェクトファイルのかわりにリストファイルが必要な場合，またはそれらのファイルのかわりにクロスリファレンスファイルが必要な場合は，ヌルデバイス（NUL）をファイル名として指定してください。次に例を示します。

```
TASM FILE1,,NUL,
```

この例ではFILE1.ASMがアセンブルされ，オブジェクトファイルFILE1.OBJとクロスリファレンスファイルFILE1.XRFが作られます。リストファイルは作成されません。

以降アセンブラプログラミングの基礎をさらに詳しく学習していきましょう。コンピュータのプログラミング言語の中でアセンブリ言語がユニークな理由もこれから説明していきます。

コンピュータのアーキテクチャ

アセンブリ言語がコンピュータが理解できる言語であると前述しましたが，その意味を理解するためにはじめにコンピュータについて詳しく学習しましょう。

つきつめて言えば，コンピュータとはデータをある場所から他の場所へ移動したり，データの論理演算または算術演算を行なう機械であるといえます。図2.2に示すように，コンピュータは入力部，制御部，演算部，メモリ，出力部の5つの機能部分（サブシステム）から構成されるシステムと考えるとわかりやすいでしょう。

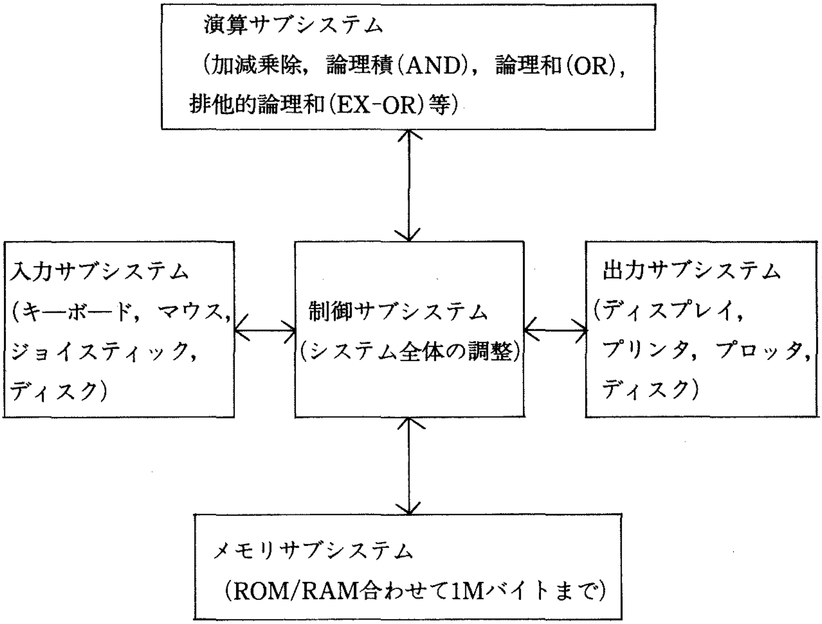


図2.2　5つのサブシステム

（ここではコンピューター一般について述べています。8088については後で簡単に触れます）。

演算サブシステムは多くの人が「コンピュータ」という言葉から連想するものです。もしコンピュータが計算機でなければいったい何でしょうか 現実にはコンピュータが計算に費やしている時間はほんのわずかで，ほとんどの時間を文字列や入出力の処理に使っています。計算のためにワードプロセッサ（ワープロ）を使う人はいません。しかし演算サブ

システムは重要で，単なる算術演算のみならず論理演算（*AND*, *OR*，**排他的***OR*など）も行ないます。

計算を行なう場合，たとえば加算を行なうためにどこから数値を持ってきて計算結果をどこへ持っていくのでしょうか？　コンピュータのメモリは高速アクセス可能な記憶装置であり，多量の文字データや数値データを処理するうえで重要な役割を果たしています。またフロッピィディスクやハードディスクは相対的にアクセス速度は落ちますが，外部メモリとして膨大な量のデータを記録することが可能です。入出力デバイス（I/O）はメモリサブシステムの一部ではなく，独立したサブシステムとなっています。

プログラムは入力サブシステムを通して，キーの押下からマウスの動きやディスク上のファイルにセーブされているデータベースに至るまで，外部からの情報を取り込むことができます。プログラムは出力サブシステムを通して，画面やプリンタにメッセージや結果を表示・印刷したり，ディスクや磁気テープにデータをセーブすることができます。ほとんどのプログラムは入出力処理を行なっています。入出力処理なくしては外部からデータを取り込むこともできなければ処理結果を表示することもできません。

制御サブシステムは他の4つのサブシステムの動作を管理し，データの動きをコントロールします。

制御サブシステムと演算サブシステムでいわゆるプロセッシングユニット（**プロセッサ CPU**）を構成します。プロセッサはコンピュータの心臓部にあたり，データ処理を行なったりメモリサブシステムや入出力サブシステムをコントロールします。プロセッサは各サブシステムの動作をコントロールしてシステムがうまく動作するように調整しているため，プロセッサがコンピュータの性能を決めると言っても過言ではありません。今日では，プロセッサ全体が1個のチップになっていることは珍しいことではありません。たとえば8088は演算部，制御部，入出力部，メモリが一体になったチップです。

プロセッサはコンピュータのハードウェアとアセンブリ言語の特有な機能を結びつけているといえます。

アセンブリ言語の特徴

前述のように，プロセッサはコンピュータの5つのサブシステムの動作の管理，数値の演算，メモリから出力デバイスへの転送などを行なっています。しかしここで基本的な疑問がわいてきます。コンピュータは必要な能力はすべて備えていますが，仕事の手順書は持っていません。ではプロセッサは処理手順を**どのように**決めるのでしょうか？

この答は意外と単純です。すなわちプロセッサはメモリからデータを取り出し，このデータによって何をすべきかを判断します。このようにプロセッサに指示を与えるデータを

命令と言います。命令はメモリに格納されたデータであり，形態は他のデータとなんら変わりありません。プロセッサが実行できる命令のセット（**命令セット**）はプロセッサのハードウェアが実行できる動作と1対1で対応しています。またソフトウェアでこれらの命令を組み合わせることによって，プロセッサにどんな処理でも行なわせることができます。たとえばプロセッサに乗算の命令がなければ，コンピュータのハードウェアが乗算を行なうことはできません。しかしソフトウェアで加算とシフト処理を行なえば，処理速度は遅くなりますが乗算を行なうことができます。ここでのキーポイントは，プロセッサの命令はコンピュータのハードウェアが本来行なうことができる動作を反映しているということです。プロセッサによって処理能力や命令セットが異なるため，すべてのアセンブリ言語は特定のプロセッサに固有の命令体系を持っています。

命令データの1つ1つはプロセッサに固有で特別な意味を持っています。たとえば命令データ4は 8088に対して次のメモリアドレスに格納されている値をALレジスタの値に加えることを指示しています(ALレジスタについてはここでは気にしないでください。後で説明します)。したがって命令データを組み合わせることによって，プロセッサに任意の処理を行なわせることができるのです。事実プログラムとは命令データを並べたものにほかなりません。

ではプロセッサはどうやって次に実行すべき命令を認識するのでしょう？　実はプロセッサは次に実行すべき命令データがあるメモリの場所を指すポインタを管理しているのです。次の命令がメモリから読み込まれて実行に移されると，ポインタは後続命令の位置を指します。命令によっては命令ポインタの値を更新するものもあります。命令をランダムに実行したり，さらには状況に応じて異なった命令の並びを実行することも可能となります。

ところでプロセッサはアセンブリ言語とどのような関係があるのでしょうか？　プロセッサの命令セットはそのプロセッサのアセンブリ言語ということができるのです。すなわちアセンブリ言語はプロセッサの命令セット（このような命令はマシンコードあるいは**機械語**として知られています)を人間に理解できる形にしたものといえます。Turbo Assemblerなどのアセンブラは，アセンブリ言語をマシンコードに翻訳します。アセンブリ言語とマシンコードは機能的には同等ですが，プログラムを作るうえではアセンブリ言語の方がはるかに簡単でしょう。たとえば

```
add  al,1
```

という命令を使う方が

```
4
1
```

というマシンコードを使うよりもプログラムが作りやすいのは明らかです。どちらも動作は同じですがアセンブリ言語を使えばマシンコード命令より憶えやすい名前（ニーモニック）で使うことができます。アセンブラはこのニーモニック命令を対応するマシンコードに翻訳するプログラムです。人間が純粋なマシンコードを理解するのは容易ではないため，このようなニーモニック命令を使うのは非常に効果的です。アセンブリ言語は，基本的にはマシンコードと1対1に対応しており，人間に理解しやすい形で使われます。

アセンブリ言語を使えば，プロセッサの1つ1つの動作をコントロールすることができ，プロセッサの性能をフルに引き出せるというメリットがあります。しかしプロセッサの1つの動作は比較的単純で，その処理範囲には限度があります。たとえば2つのlong型の整数を加算してその結果をlong型の整数変数に代入する場合は，C言語では次の1行で済みます。

```
i = j + k;
```

一方8088アセンブラでは次のように6命令となります。

```
mov   ax,[j]
mov   dx,[j+2]
add   ax,[k]
adc   dx,[k+2]
mov   [i],ax
mov   [i+2],dx
```

もちろんC言語のコードはコンパイルされてアセンブラ命令に対応した6つのマシンコード命令に翻訳されます。しかしアセンブラの6命令よりもC言語の1行を書く方が簡単です（アセンブラ命令はコンピュータの基本的な性能を反映します。いかなる言語で書かれたプログラムでも，最終的にマシンコードに翻訳しないと実行することはできません）。

ではなぜ他の言語より難しいアセンブラを使うのでしょうか？　理由の1つに，アセンブリ言語プログラムはプロセッサができることなら何でも指定できることにあります。そのためアセンブラを使えば任意のメモリ空間にアクセスしたり，入出力デバイスを直接コントロールすることができます。もう1つの理由として，アセンブリ言語はコンピュータが直接理解できる言語であることが挙げられます。このことは，よくできたアセンブラプログラムが一番処理速度が速いことからもわかります。他の言語で書かれたプログラムの場合はマシンコードに翻訳しなければならないため，プログラムの品質は翻訳のよしあしに左右されます。しかしアセンブリ言語の場合は直接マシンコードに置き換わるため，非常に効率よく翻訳されます。したがってアセンブリ言語を使えば，プログラムの指示どおりにコンピュータに仕事をさせることができます。

もちろんアセンブリ言語といえども効率の悪いプログラムでは処理速度は上がりません。プロセッサはプログラムの指示どおり**忠実に**命令を実行しているだけだからです。またアセンブリ言語はデータ型の変換やミス（変数の上書き，配列の終端を越えるなど）の防止

策をほとんどサポートしていません。したがってアセンブリ言語でプログラムを作る場合は，他の言語で作る場合よりいっそうの注意力とプログラミング能力がプログラマに要求されます。

これでプロセッサとアセンブリ言語の互いの関連についての説明を終わり，引き続き8088のアセンブリ言語について説明します。

8088と8086プロセッサ

8088は世界中でもっともポピュラーな機種であるIBM PCとXTに使われているプロセッサです。しかし8088はiAPx86ファミリとして知られたプロセッサファミリの中の1つのプロセッサに過ぎず，このファミリには8086, 80286, 80386などが含まれます。それぞれのプロセッサは以下の機種で使用されています。

8086	NEC PC-9801シリーズのE, F, M（VF, VMではNEC製の同等品V30を使用） IBMモデル25, 30
80286	NEC PC-9801シリーズのXA, XL, VX, UX, RX IBM AT, IBM PS/2モデル50, 60
80386	NEC PC-9801シリーズのXL2, RA, RL IBM PS/2モデル80

これらのプロセッサは8088とは仕様が異なっています。第11章「80386と他のプロセッサ」ではiAPx86ファミリのいろいろな製品について解説します。iAPx86ファミリのすべての製品に共通することは8086や8088プロセッサ用のプログラムが使用できることです。

8086はiAPx86ファミリの ルーツ にあたるプロセッサです。8088は8086にスケールダウンした外部データバスを付けたものです。メモリとプロセッサ間で転送されるデータは8086では16ビット，8088では8ビットです。これら2つのプロセッサはまったく同じ命令セットを持っています。したがってNEC PC-9801シリーズやIBM PCおよびそれらの後継機上で走るプログラムは8088用のアセンブリ言語ではなく，8086用のアセンブリ言語で書かれています。以後このマニュアルでは8086アセンブリ言語が8088アセンブリ言語をカバーしていることを前提として説明を進めます。

8086の性能

今日の基準では8086は中程度の性能をもつプロセッサです。8086は10年前にデザインされたものです。10年間のテクノロジーの進歩にはめざましいものがあり，チップデザインの分野にも大きな技術革新をもたらしました。にもかかわらず8086は依然として主要なプロセッサしての位置付けられています。この理由の1つに，NEC PC-9801シリーズやIBM PCおよびそれらの互換機の数が膨大なことがあります。特にIBM PCは1千万台以上普及し

ており，そのようなコンピュータの存在を無視することはできません。他の理由に8086は今日でも高度なソフトウェアのニーズに応えていることです。

8086の特長として，アクセス可能なメモリ空間が大きいこと（100万文字または100万バイトに相当する空間），命令セットが強力であること，性能のよいソフトウェアをサポートできることなどが挙げられます。しかし8086といえども超高速プロセッサではなく，すべての言語の能力を引き出せるわけではありません。8086用の優れたプログラムを作るためには，アセンブリ言語にまさるプログラミング言語は他にないでしょう（8086は4.77または8MHz, 80286は6, 8, 10, 12および16MHz, 80386は16, 20, 25MHzで動作します）。

アセンブリ言語プログラマが8086を通じて使用できる資源は，メモリ，入出力（I/O）インターフェース，レジスタ，そしてもちろん命令です。次にこれらの資源について解説します。

メモリ

8086は1Mバイトのメモリ空間にアクセスすることができます（1Mバイトとは2の20乗つまり1,048,576バイトを指します。1バイトは8ビットです）。メモリの先頭バイトはアドレス0にあたり，最終バイトはアドレス0FFFFFhにあたります（次のページの図2.3を参照してください）。

（最終アドレスの0FFFFFhは16進数です。16進数は最後にhを付けて表記します。最終アドレスのこの値は10進数の1,048,575にあたります）。アセンブリ言語でプログラムを作るためにはこの16進数にぜひ慣れておきましょう。16進数については第4章「アセンブラプログラミングの基礎」で説明します。

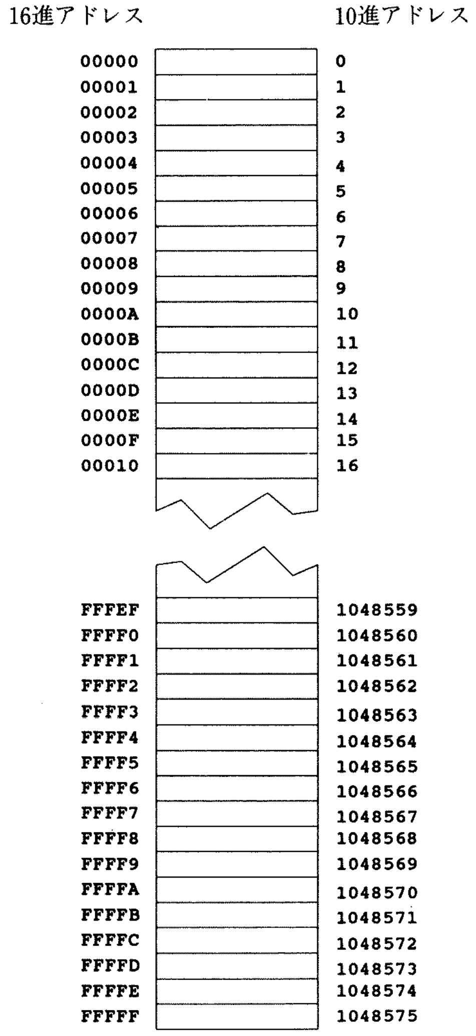


図2.3 8086のメモリアドレス

1バイト（8ビット）は1文字または0～255の範囲の整数を表わします。8086はバイトよりも大きな値も扱うこともできます。2バイト（**ワード**と言います）を使えば0～65,535の範囲の整数を扱うことができ，8086はワードもバイトと同様な速度で処理することができます。

4バイト（**ダブルワード**）を使えば0～4,294,967,295の範囲の整数，すなわち単精度浮動小数を扱うことができます。8バイト（**クオッドワード**）を使えば倍精度浮動小数を扱うことができます。8086は最後の2つのデータ型は直接扱うことはできませんが，8087という数値

演算コプロセッサを使えば，浮動小数や高精度の整数を直接扱うことができます。また適切なソフトウェアさえ使えば，速度は遅くなりますが実質的に8086でどんな型のデータでも扱うことが可能となります。

8086はプログラムは1,000,000バイト以上のメモリの任意のアドレスにアクセスすることができます。たとえば次のプログラムはアドレス0の内容をALレジスタにロードします。

```
.
.
.
mov   ax,0
mov   ds,ax
mov   bx,0
mov   al,[bx]
.
.
.
```

ここでは詳細は気にしないでください。8086は1,000,000バイト以上のメモリ空間にアクセスすることができ，このメモリ内のデータを高速かつ柔軟に読み書きできるということがポイントです。

1Mバイト（メガバイト）は64Kバイトよりはるかに大きいメモリ容量です（64Kバイトは2の16乗つまり65,536バイトです)。一方8086ファミリの最新チップである80386は8086の4000倍のメモリ空間を扱うことができます。したがって8086が扱えるメモリ空間はかなり小さいと言えましょう。通常NEC PC-9801やIBM PCにMS-DOSを載せている場合は，640Kバイトしか汎用メモリとして使用できません。残りの部分はシステムソフトウェア用および画面用メモリとして使われています。命令はデータと同様にメモリに格納されており，プログラムコードもデータも640Kバイトの汎用メモリ内に格納しなければなりません。

8086には1Mバイトのメモリをアドレシングする能力があるにもかかわらず，64Kバイトのを越えるメモリにアクセスすることは容易ではありません。これは8086には**セグメント**と呼ばれる概念があるためです。セグメントについては少し後の「セグメントレジスタ」の節で説明します。

入力と出力

8086は入出力デバイスを2つの方法でサポートします。1つは入出力（I/O）命令で，もう1つはメモリアドレスです。入出力デバイスの中にはポートによってコントロールされるものもあります。ポートとは，図2.4に示すように64Kバイトのメモリ中の特別なアドレスを指します。1Mバイトのメモリアドレスとは別のものです。

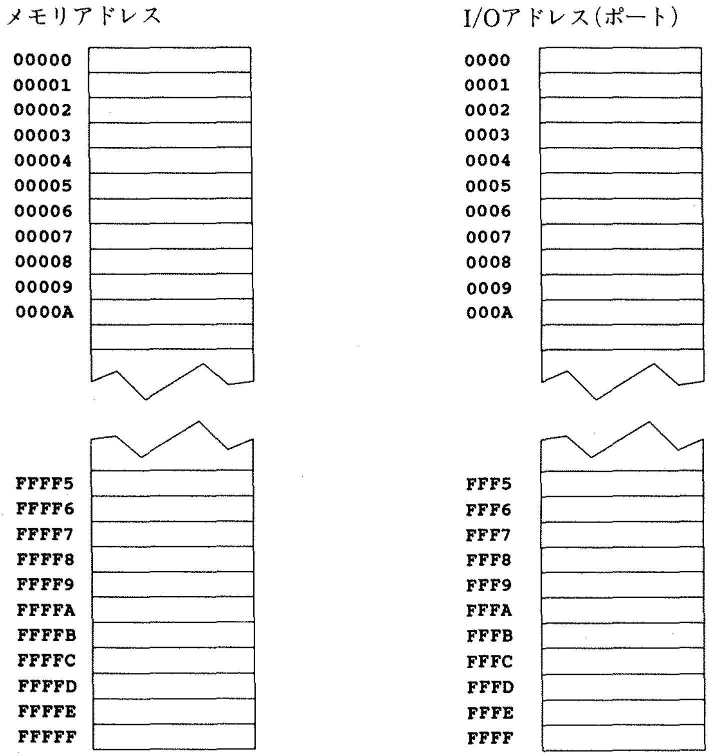


図2.4 8086のメモリアドレスとI/Oアドレスの分離

8086のI/Oアドレスの数は，メモリアドレスに比べると非常に少なくなっています。IBM PCについて言えば，技術的にはI/Oアドレスとして64Kバイト分とることが可能ですが，実際にはわずか4Kバイトしかありません。つまりI/Oアドレスはデータを保存するためにあるのではなく，入出力デバイスに対するコントロールチャネルとデータチャネルを提供するためにあるのです。たとえばモデムなどのシリアルデバイスはいくつかのI/Oアドレスによってコントロールされます。 I/OアドレスはINとOUTというそれぞれ入力と出力専用の命令によってのみアクセスされます。たとえば次の例，はALレジスタの内容をDXレジスタで指定されたI/Oポートへ出力します。

```
out dx,al
```

I/O命令とI/O一般については第4章「アセンブラプログラミングの基礎」で説明します。出力デバイスによってはI/O命令ではなくメモリアドレスによってコントロールされるもの

もあります。この手法は**メモリマップド**I/Oと言われます。この方法はメモリ空間の16K, 32K, あるいは256Kバイトを使う**ビットマップ**方式（画面上のドットの並びがビットの並びに対応する）のディスプレイでよく使われます。

どんなデバイスでもI/Oまたはメモリマップアドレスによってコントロールすることが可能です。実際ほとんどのディスプレイでは，機能によってI/O命令とメモリアドレスのどちらかに応答するようになっています。

レジスタ

8086には**レジスタ**と呼ばれる高速な一時記憶がいくつかあります。レジスタは通常のメモリより高速でアクセスできる特別なメモリと考えることもできます。それぞれのレジスタは固有の特長を備え，他のレジスタやメモリがサポートしていない機能を持っています。

レジスタは図2.5に示すように次の4種類に分類することができます。

フラグレジスタ
汎用レジスタ
命令ポインタ
セグメントレジスタ

ではこれらのレジスタについて見ていきましょう。

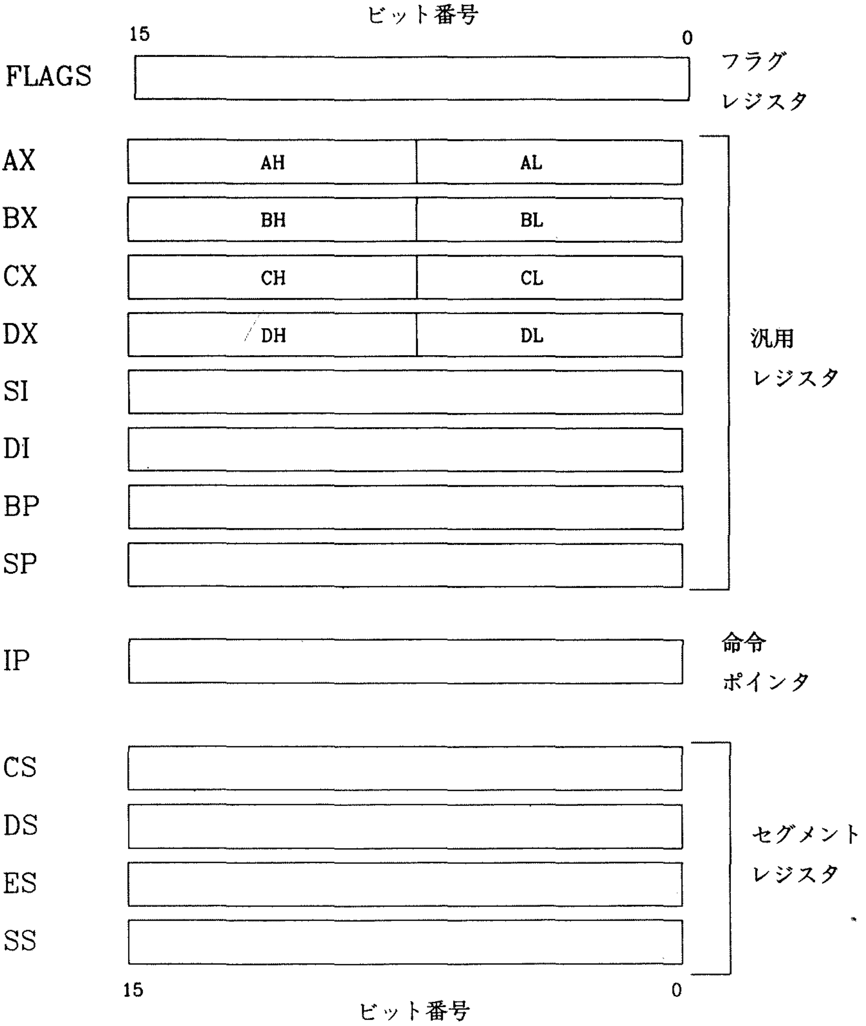


図2.5 8086のレジスタ構成

フラグレジスタ

16ビットのフラグレジスタは図2.6に示すように，8086の状態に関する情報と最後に実行した命令の結果を保持します。

ビット番号

15															0
				O	D	I	T	S	Z		A		P		C

フラグ ビット

O：オーバーフローフラグ
D：ディレクション(方向)フラグ
I：インタラプト(割り込み)フラグ
T：トラップフラグ
S：サイン(符号)フラグ
Z：ゼロフラグ
A：補助キャリフラグ
P：パリティフラグ
C：キャリフラグ

図2.6 8086のフラグレジスタ

たとえば減算の結果がゼロかどうかを知りたい場合は，命令実行の直後に**ゼロフラグ**(フラグレジスタのZビット）をチェックします。Zビットがセットされていた場合は減算の結果がゼロであったことがわかります。**キャリ，オーバフローなどの他のフラグ**も算術演算や論理演算の結果を示すものです。

それ以外のビットは，8086の動作のコントロールモードを示します。**ディレクションフラグ**は文字列転送等の方向を示します。**インタラプト（割り込み）フラグ**は，キーボードやモデムなどの外部装置がコンピュータの処理を一時的に中断し，他の緊急処理を行なえるようにできるか否かをします。**トラップフラグ**はソフトウェアで他のソフトウェアをデバッグするときなどで使われます。

フラグレジスタは変更したり読み出しすことはできません。その場合は通常(**CLD, STI, CMC**などの）特別な命令を使ってコントロールするか，算術演算や論理演算によってフラグをコントロールします。同様にフラグレジスタの特定ビットの状態によって**JZ, RCR, MOVSB**などの命令の実行結果が異なります。フラグレジスタはメモリとして使われるのではなく，8086の状態をチェックしたりコントロールするために使われます。

他のレジスタやメモリにはデータを格納することができます。フラグレジスタはデータ間の関係，処理結果，8086自身の状態などに関する情報を表わします。

汎用レジスタ

8086の8個の汎用レジスタ（すべて16ビット）は計算やデータ転送の際の転送元と転送先，メモリのポインタ，カウンタとしてほとんどの命令に関係しています。それぞれの汎

用レジスタには16ビットのデータをストアでき，メモリ間でのデータ転送，算術演算や論理演算に使うことができます。たとえば次のプログラムはAXレジスタに5を，DXレジスタに9をそれぞれセットし，2つの値を加えてその結果14をAXレジスタに再びセットします。

```
.
.
.
mov  ax,5
mov  dx,9
add  ax,dx
.
.
.
```

この例ではAXレジスタやDXレジスタのかわりに，CX, SIまたは他の汎用レジスタを使っても同じ結果を得ることができます。

汎用レジスタは値を保持したりデータ処理の際の転送元と転送先を表わすだけでなく，それぞれ固有の機能を持っています。次にそれぞれの汎用レジスタについて説明します。

AXレジスタ

AXレジスタは**アキュムレータ（累積器）**としても知られています。乗算や除算を実行する際は必らずこのAXレジスタが関係します。また算術演算や論理演算，データ転送の際にもっとも役に立つレジスタです。

AXレジスタの下位8ビットはALレジスタと呼ばれ，上位8ビットはAHレジスタと呼ばれます（ALはA-Low, AHはA-Highの意味です）。このようにAXレジスタは2つの別々のレジスタとして扱えるため，バイト単位でデータを扱うときに便利です。次のプログラムではAHに0をセットし，ALにその値をコピーしALに1を加えています。

```
.
.
.
mov  ah,0
mov  al,ah
inc  al
.
.
.
```

この結果AXレジスタに1がセットされます。同様にBX, CX, DXの各レジスタも，16ビットレジスタとしても，2つの8ビットレジスタとしても使うことができます。

BXレジスタ

BXレジスタはメモリアドレスを指定することができます。詳細は第4章「アセンブラプログラミングの基礎」で説明します。簡単に言うと，BXレジスタにストアされた16ビットの値をメモリアドレスの一部として使うこともできるということです。たとえば次のプログラムではメモリアドレスの9番地の内容をALレジスタにロードします。

```
 .
 .
 .
mov  ax,0
mov  ds,ax
mov  bx,9
mov  al,[bx]
 .
 .
 .
```

この例では，メモリにアクセスする前にAXレジスタによってDSレジスタに0をセットしています。これは前述した8086のセグメントに関係する処理です。セグメントについては，少し後の「セグメントレジスタ」の節で説明します。特に指定がない場合，BXをメモリポインタとして使うと，その値はDSセグメントレジスタからの相対アドレスを示します。 AX, CX, DXレジスタと同様，BXレジスタもBHとBLの2つの8ビットレジスタとして扱うことができます。

CXレジスタ

CXレジスタの特徴はカウンタとして使われることです。たとえば次のプログラムでは，CXレジスタを使って命令ブロックを10回繰り返しています。

```
    .
    .
    .
    mov   cx,10
BeginningOfLoop:
    .
    .
    .
    <繰り返し実行する命令>
    .
    .
    .
    sub   cx,1
    jnz   BeginningOfLoop
    .
    .
    .
```

プログラムの中に見慣れないものがあっても気にしないでください。ここでのポイントは，ラベル*BeginningOfLoop*と**JNZ**命令の間にある命令が，CXレジスタが0になるまで繰り返されるということです。この例では，CXレジスタが0でないかぎりCXをカウントダウンして*BeginningOfLoop*に戻るために，2つの命令（**sub cx, 1**と**jnz** *BeginningOfLoop*）が必要となることに注意してください。

カウントダウンとループはプログラムで頻繁に使われる手法です。8086にはループ処理を高速かつコンパクトにするための特別な命令，**LOOP**が用意されています。この**LOOP**命令1つで，CXレジスタから1を引きCXが0でない限りジャンプするという一連の処理を行なうことができます。さきほどのプログラムを**LOOP**命令を使って書き直すと次のようになります。

```
    .
    .
    .
    mov   cx,10
BeginningOfLoop:
    .
    .
    .
    <繰り返し実行する命令>
    .
    .
    .
    loop  BeginningOfLoop
    .
    .
    .
```

ループについては第4章「アセンブラプログラミングの基礎」で説明します。ここではCXレジスタはカウントとループ処理に有用であることを憶えておいてください。

AX, BX, DXレジスタと同様，CXレジスタもCHとCLの2つの8ビットレジスタとして扱うことができます。

DXレジスタ

DXレジスタは**IN**命令や**OUT**命令とともにI/Oアドレスポインタとして使えるただ1つのレジスタです。DXレジスタを使わずにI/Oポートの256～65, 535にアクセスすることは不可能です。たとえば次のプログラムではI/Oポート1000に62を書き込みます。

```
.
.
.
mov  al,62
mov  dx,1000
out  dx,al
.
.
.
```

DXレジスタの他の特徴は除算と乗算に関係することです。32ビットの分子を16ビットの分母で割る場合，分子の上位16ビットはDXに入っていなければなりません。計算後には余りがDXにストアされます（分子の下位16ビットはAXに入っていなければなりません。商がAXにストアされます)。同様に2つの16ビット値を掛けると，積の上位16ビットはDXに，下位16ビットはAXにそれぞれストアされます。

AX, BX, CXレジスタと同様，DXレジスタもDHとDLの2つの8ビットレジスタとして扱うことができます。

SIレジスタ

BXレジスタと同様，SIレジスタもメモリポインタとして使うことができます。たとえば次のプログラムでは，メモリの20番地にある8ビットの値をALレジスタにロードします。

```
.
.
.
mov  ax,0
mov  ds,ax
mov  si,20
mov  al,[si]
.
.
.
```

SIレジスタを8086の文字列命令とともに使うと，強力なメモリポインタになります。たとえば次のプログラムではSIで指定されるメモリ内容をALにロードした後,SIの内容をインクリメントします。

```
.
.
.
cld
mov  ax,0
mov  ds,ax
mov  si,20
lodsb
.
.
.
```

この手法は文字列などを扱う際に，メモリの連続アドレスにアクセスする場合に非常に効果的です。さらに文字列命令は任意の回数繰り返すことが可能で，1つの命令で数百，数千回の処理を繰り返し実行することができます。文字列命令については第5章「Turbo Assemblerプログラミング入門」で説明します。

DIレジスタ

DIレジスタもメモリポインタとして使うことができます。強力な文字列命令とともに使うと重宝するという点でSIレジスタによく似ています。たとえば次のプログラムでは1024番地の8ビットの値とBLレジスタの値を加えます。

```
.
.
.
mov   ax,0
mov   ds,ax
mov   di,1024
add   bl,[di]
.
.
.
```

文字列命令に関してはDIレジスタの役割はSIレジスタのそれと若干異なっています。SIレジスタは常に転送元のメモリポインタとして使われますが，DIレジスタは常に転送先のメモリポインタとして使われます。またSIは通常DSセグメントレジスタのメモリアドレスからの相対アドレスを示しますが，DIはESセグメントレジスタのメモリアドレスからの相対アドレスを示します(SIとDIを文字列命令以外の命令で使うと，常にDSのメモリアドレスからの相対アドレスを示します)。たとえば次のプログラムは，**STOSB**文字列命令によって，DIの指定するメモリアドレスにALの値を格納して，DIの値をインクリメントします。

```
.
.
.
cld
mov   dx,0
mov   es,dx
mov   di,2048
stosb
.
.
.
```

文字列命令を使うにはセグメントとセグメントレジスタの知識が必要です。文字列命令については第5章「Turbo Assemblerプログラミング入門」で説明します。

BPレジスタ

BX, SI, DIレジスタと同様，BPレジスタもメモリポインタとして使うことができます。ただしBX, SI, DIの各レジスタは通常DSセグメントレジスタのメモリアドレスからの相対アドレスを示すの対して(文字列命令とともに使うとDIレジスタはESセグメントレジスタのメモリアドレスからの相対アドレスを示す)，BPレジスタはSS (**スタックセグメント**) レジスタのメモリアドレスからの相対アドレスを示します。

セグメントについてはまだ説明していませんが，基本概念は後述します。パラメータをサブルーチンにわたす最も効率の良い方法はパラメータをスタックに入れる (プッシュ)

することです。C言語とPascalではこの方法をとっています。C言語がスタックを使う理由とその方法については，第6章「Turbo AssemblerとTurbo Cのインターフェース」を参照してください。

スタックはSSレジスタが示すセグメント内にあり，データは通常DS(データセグメント)レジスタが示すセグメント内にあります。BX, SI, DIの各レジスタは通常データセグメントを指すため，これらのレジスタを使ってもスタックに渡されたパラメータを指すことはできません。なぜならば，通常スタックはスタックセグメントという別のセグメントに存在するからです。

BPレジスタを使えばスタックセグメント内のアドレスにアクセスすることができ，この問題は解決できます。たとえば次のプログラムではスタックセグメントにアクセスして，Turbo Cからアセンブラのサブルーチンに渡す第1引数をAXレジスタにロードしています。

```
.
.
.
push   bp
mov    bp,sp
mov    ax,[bp+4]
.
.
.
```

簡単にいうと，BPレジスタの機能はパラメータやローカル変数の他，スタック経由のメモリアクセスをサポートすることです。

SPレジスタ

SPレジスタは**スタックポインタ**として知られていますが，汎用レジスタの中では最も特殊な用途に使われます。通常このレジスタはスタックの管理専用に使われます。スタックとはLIFO（後入れ先出し）方式でデータを出し入れするメモリ領域です。スタックに最後にストアされたデータは最初に読み出されます。従来からスタックを説明するために，積み上げられた皿（以後〝皿のスタック〟と呼びます）がたとえとして引用されます。この皿のスタックに新しい皿を加えるには，一番上に積まなければなりません。また皿を取り出すには，やはり一番上から持っていかなければなりません。したがって，最初にスタックに積まれた皿は，最後にしか取り出すことができないことになります。

SPレジスタはいかなるときでもスタック領域の一番上のアドレスを指しています。皿のスタックで考えればわかるように，スタックの一番上とは次の値をストアできる位置のことです。スタックにデータをストアすることを**プッシュ**と言い，このための命令が**PUSH**です。反対にスタックからデータを取り出すことを**ポップ**と言い，そのための命令が**POP**

です。

図2.7は，次のプログラムを実行するとSP, AX, BXの各レジスタがどのように変化するかを表わしています（SPの初期値は1,000です)。

```
.
.
.
mov  ax,1
push ax
mov  bx,2
push bx
pop  ax
pop  bx
.
.
.
```

他の汎用レジスタと同様，SPに対して値のストア，加算，減算を行なうことができます。ただし，SPに関する操作を行なう際はよくよく気をつけなければなりません。SPの値を変えることはスタックの一番上の位置を変更することになり，致命的なエラーが発生する可能性が大きくなります。

スタックに関する操作はPUSHやPOPがすべてではありません。サブルーチン，手続き，関数のコールおよびリターンの際は必らずスタックが使われます。またキーボードやシステムクロックなど，デバイスによっては8086に割り込みをかける際にスタックを使う場合があります。つまりスタックはいつ使われるかわからないということです。したがってたとえ一瞬でもSPを変更すると，他のデバイスが必要とするときに正しいスタックを使えないという事態が起きてしまいます。

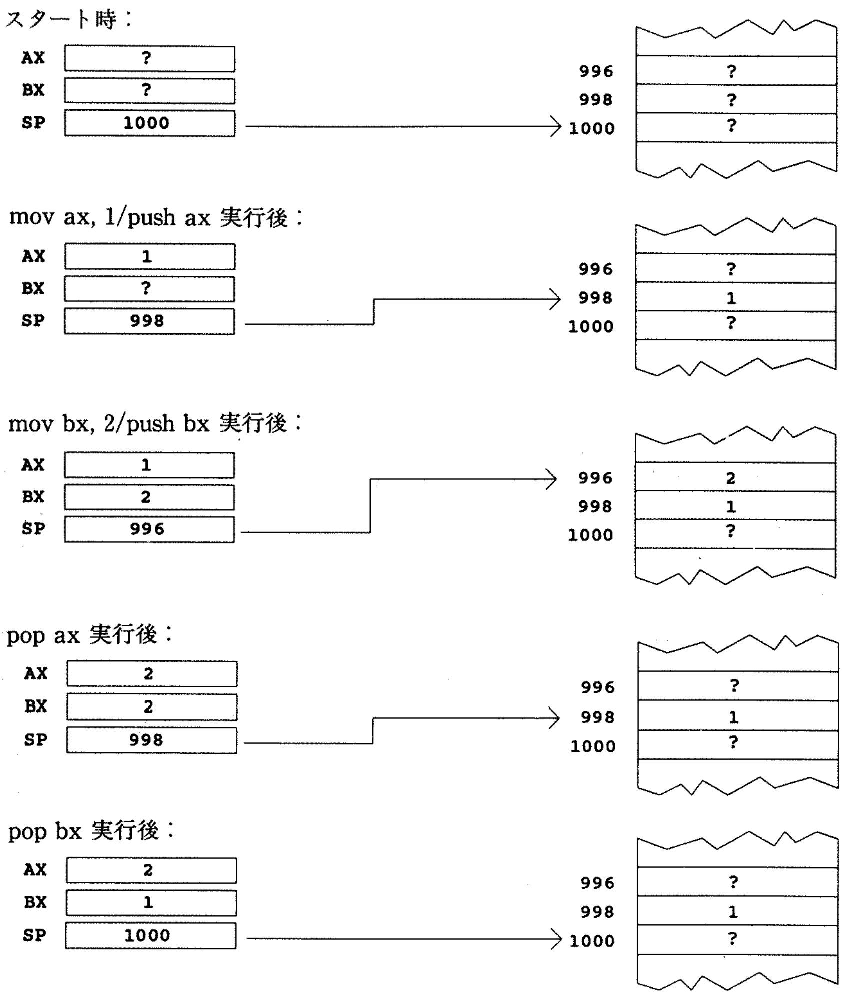


図 2.7 AX, BX, SPとスタックの関係

動作に確信が持てないときはスタックを操作してはなりません。プッシュ，ポップ，コール，リターンは自由に使ってかまいませんが，SPの値を直接変更することは避けてください。他の7つの汎用レジスタはいつでも変更してもかまいません。

命令ポインタ

命令ポインタ（IP）は，次に実行する命令がストアされているメモリオフセットを示し

ます。1つの命令が実行されると命令ポインタが更新され，次の命令のメモリアドレスを指します。通常は次のメモリアドレスにある命令が次に実行すべき命令ですが，CALLやJUMPなどの命令の後には新たな値が命令ポインタにセットされてプログラムの他の場所にジャンプします。

命令ポインタを直接読み出すことはできません。上記のCALLやJUMPなどの分岐命令によってのみ命令ポインタの内容を変更することができます。

命令ポインタ単独で次に実行すべき命令のメモリアドレスを指し示すわけではありません。前にも述べたように，8086のメモリアドレスはセグメント方式を採用しているため，アドレシングは若干複雑になっています。命令フェッチの際はCSセグメントレジスタがベースアドレスを指定し，命令ポインタがこのベースアドレスからのオフセットを指定します。

メモリのアドレシングについて説明するときは常にセグメントについて触れてきましたが，いつも中途半端な説明で終っていました。次にセグメントについて詳しく説明します。

セグメントレジスタ

次に8086の最大の特徴であるメモリのセグメントについてお話ししましょう。セグメントの基本概念は次のとおりです。8086は1Mバイトのメモリ空間をアドレシングすることができますが，本来これだけのメモリ空間を表わすためには20ビットが必要となります。ところが8086の命令ポインタは16ビットしかありません。前に説明したBXレジスタなどの汎用レジスタはすべて16ビット構成でした。では8086はこれらの16ビットレジスタを使ってどのように20ビットのメモリアドレスを扱っているのでしょうか。

8086は実は20ビットのメモリアドレスを2つの要素にわけているのです。確かに16ビットポインタはそのまま利用しますが，これらのポインタはメモリアドレスの一部（すなわちオフセット）を示すに過ぎません。この**オフセット**と16ビットセグメントレジスタとを組み合わせて20ビットのメモリアドレスを表わしているのです。

セグメントとオフセットの関係は次のようになります。セグメントレジスタの値は4ビット分左にシフトされ（つまり16倍され）オフセットの値に加えられます（図2.8を参照してください）。

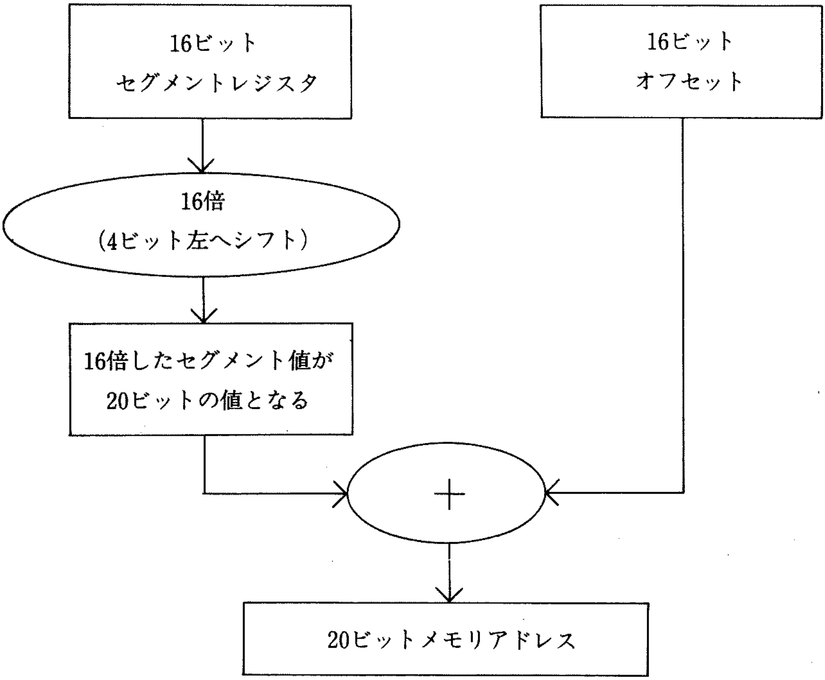


図2.8　20ビットメモリアドレス

次に例を示します。

```
.
.
.
mov  ax,1000h
mov  ds,ax
mov  si,201h
mov  dl,[si]
.
.
.
```

この例ではDSセグメントレジスタは1000hに，SIレジスタは201hにそれぞれセットされています。これによって1000:201hというセグメントとオフセットのペアができます（セグメント オフセットの計算は16進法でのみ行なうことができます。したがって16進数に慣れておくのがよいでしょう）。DLレジスタにはアドレス（(*DS*＊*16*)＋*SI*）つまり（(*1000h*＊*16*)＋*201h*）の内容がセットされます。

$$\begin{array}{r} 1000h \\ \times \quad 16 \\ \hline 10000h \\ +\ 201h \\ \hline 10201h \end{array}$$

図2.9にその例を示します。

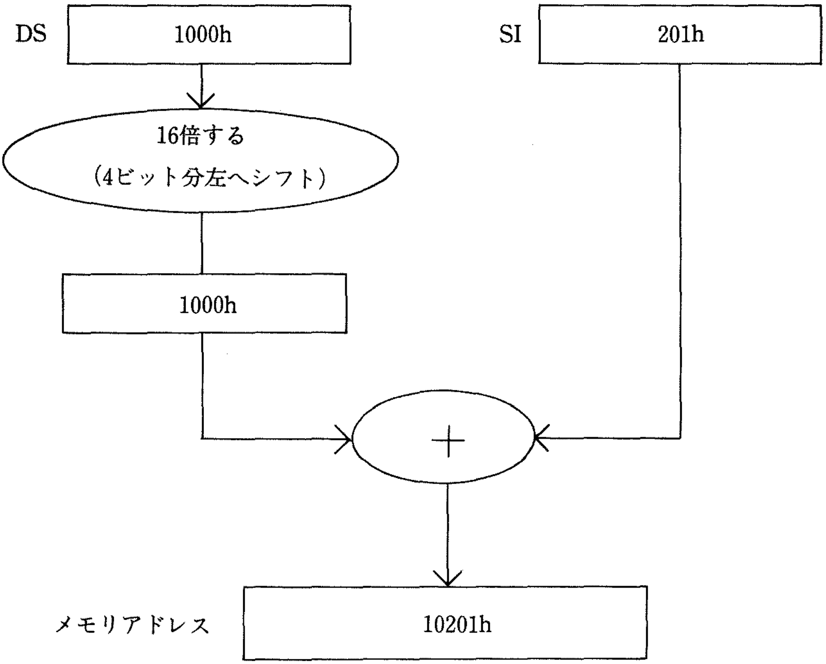


図2.9　mov命令によるメモリアドレスの計算

セグメントのもう1つの計算方法は，セグメントレジスタの値を左へ4ビットつまり16進数の1桁分シフトすることです。これは16を掛けるのと同じことです。

$$\begin{array}{r} 10000 \\ +\quad 201 \\ \hline 10201 \end{array}$$

プログラムはこのセグメント：オフセットのペアにしたがってのみ，8086の1Mバイトのメモリ空間にアクセスすることができます。プログラマはメモリにアクセスするときは**常に**セグメント：オフセットのペアを指定しなければなりません。8086のすべての命令やアドレシングモードは，いずれかのセグメントレジスタの値に関係しています。命令によってはセグメントレジスタを必要に応じて指定できるものもあります。

プログラマはセグメントレジスタに値をセットすることはほとんどありません。そのかわりセグメントレジスタにセグメント名をセットすることがあります。このセグメント名は，プログラムのアセンブル，リンク，実行という一連のプロセスで数値に変換されます。このような処理が必要となる理由は，あるセグメントがメモリのどこにあるか前もって知る方法が無いからです。メモリ内の場所は，DOSのバージョン，メモリ常駐のプログラムの数とサイズ，残りのプログラムが必要とするメモリ容量などによって異なってきます。セグメント名を使うことによって，Turbo AssemblerとDOSはその機能をフルに発揮することができます。

もっともよく使われるセグメント名は**@data**です。このセグメント名は簡略セグメント疑似命令が使われたときはデフォルトのデータセグメントを示します。たとえば次のプログラムでは，DSセグメントレジスタに*Var1*が存在するデフォルトデータセグメントのポインタをセットしています。

```
      .
      .
      .
      DOSSEG
      .MODEL SMALL
      .DATA
Var1 DW 0
      .
      .
      .
      .CODE
      mov  ax,@data
      mov  ds,ax
      .
      .
      .
      END
```

次の章では簡略セグメント疑似命令とセグメントレジスタのセットについて説明します。8086でセグメントを使うことは興味あることです。16ビットの値で指定できるメモリブロックは最大64Kであるため，あるセグメントレジスタの値を基に一たびにアドレシングできるメモリオフセットはわずか64Kバイトしかありません。セグメントレジスタとオフセット値は頻繁に変更されるため，8086で64Kバイトを越えるデータを扱うことは難しくなります。

8086の大容量のメモリをアドレシングすることは容易ではありません。その理由はセグメントレジスタは汎用レジスタと違って，算術演算命令や論理演算命令の転送元や転送先として使うことができないからです。現実にセグメントレジスタと汎用レジスタ間あるいはセグメントレジスタとメモリ間でのデータ転送は，セグメントレジスタを使ってのみ実行することができます。たとえばESレジスタに100を加えるには次のような処理が必要となります。

```
.
.
.
mov  ax,es
add  ax,100
mov  es,ax
.
.
.
```

ここまでの説明から，8086は64Kバイト以内のメモリブロックを扱うことを得意としていることがわかります。

セグメントを使うもう1つの理由に，セグメントオフセットのペアを使えば工夫しだいで任意のメモリアドレスをアドレシングできることが挙げられます。たとえばメモリアドレスの100hは0：100h, 1：F0h, 2：E0hなどのセグメントオフセットのペアによってアドレシングすることができます。これらのセグメント：オフセットのペアはいずれもアドレス100hを指定します。

汎用レジスタと同様に，各セグメントレジスタの役割は決っています。CSレジスタはプログラムコード，DSレジスタはデータ，SSレジスタはスタック，ESレジスタはワイルドカード（エクストラ）セグメントをそれぞれ指定します。ESレジスタは必要に応じて任意のセグメントを指定することができます。では各セグメントレジスタについて説明します。

CSレジスタ

CSレジスタは64Kバイトの**コードセグメント**のメモリブロックを指定します。コードセグメントの中には次に実行すべき命令がストアされています。次に実行する命令はコードセグメント中のIPで指定されるオフセットにあります。この場合のセグメント：オフセットペアはCS：IPとなります。8086はCS以外で指定されたセグメントから命令をフェッチすることはありえません。

CSレジスタはJUMP, CALL, RETURNなどの多くの命令によって変更されます。いかなる場合でも，CSレジスタの内容を直接セットすることはできません。

一般的にメモリアドレシングモードやメモリポインタ（IP以外）によってCSに関連する

処理を行なうことはできません。

DSレジスタ

DSレジスタはデータセグメントの先頭位置を指定します。データセグメントは64Kバイトのメモリブロックで，ほとんどのメモリオペランドはこのセグメント内にストアされます。通常BX, SI, DIなどが関係するメモリオフセットの処理は，ダイレクトメモリアクセスなどと同様にDSレジスタと関連しています。データセグメントは基本的にはその名が示すとおり，カレントデータセットが存在するセグメントです。メモリアドレスの詳細は第4章「アセンブラプログラミングの基礎」で説明します。

ESレジスタ

ESレジスタは**エクストラセグメント**として知られる64Kバイトのメモリブロックを指定します。その名が示すように，エクストラセグメントは特定の目的をもっているわけではありません。必要に応じてどんな用途にも使うことができます。エクストラセグメントはデータ格納のために新たな64Kバイトのメモリブロックを設けるために使われたりします。しかし通常はエクストラセグメント内でのメモリアクセスはデータセグメント内でのそれよりも効率が悪くなっています。詳細は第10章「Turbo Assemblerの応用プログラミング」で説明します。

エクストラセグメントがその威力を発揮するのは，文字列命令を使う場合です。文字列命令によってメモリにデータを書き込む場合は，常にES：DIによって目的のメモリアドレスを指定します。したがってブロックコピー，文字列比較，メモリスキャン，メモリブロックのクリアなどの際に，エクストラセグメントを転送先のセグメントとして利用すると非常に便利です。文字列命令とそれに関連したESレジスタの使いかたについては，第5章「Turbo Assemblerプログラミング入門」で説明します。

SSレジスタ

SSレジスタはスタックセグメントの先頭を指定します。スタックセグメントは64Kバイトのメモリブロックでスタックが存在するセグメントです。SPレジスタに影響を与える命令（PUSH, POP, CALL, RETURNなど）はすべてスタックセグメントを利用します。なぜならばSPはスタックセグメント内のメモリにのみアドレシングすることができるからです。　すでに説明したように，BPレジスタもスタックセグメントに関連しています。BPレジスタはスタックにストアされているパラメータや変数にアクセスするために使うことが

できます。メモリアドレスの詳細は次の章で説明します。

8086の命令セット

プログラマにとって8086の命令セットは非常に有益なものです。前述のとおり，命令セットの中にはプログラマが8086に実行させたいすべての命令が含まれています。Turbo Assemblerの命令セットの一覧はこの後の表2.1を参照してください。

ご存じのように8086の命令セットには多くの命令があります。これらの命令の中は何の処理も行なわない**NOP**命令から最大65, 535バイトをコピーする**REP MOVSB**命令に至るまで，多種多様な命令が含まれます。この章の残りの部分と第4章，第5章，第10章で8086の命令セットについて詳しく説明します。

表2.1　Turbo Assemblerの命令セット

命令	プロセッサ	命令	プロセッサ
ADD	全部	SAL	全部
OR	全部	SHL	全部
ADC	全部	SAR	全部
SBB	全部	SHR	全部
AND	全部	RCL	全部
SUB	全部	RCR	全部
XOR	全部	AAA	全部
CMP	全部	AAS	全部
XCHG	全部	CBW	全部
TEST	全部	CWDE	386
MOV	全部[1]	CLC	全部
ESC	全部	CLD	全部
JMP	全部	CLI	全部
CALL	全部	CMC	全部
INT	全部	CMPSB	全部
INC	全部	CMPSW	全部
DEC	全部	CMPSD	386
PUSH	全部	CWD	全部
POP	全部	CDQ	386
AAD	全部[2]	DAA	全部
AAM	全部[2]	DAS	全部
IN	全部	HLT	全部
OUT	全部	INTO	全部
LEA	全部	IRET	全部
LDS	全部	IRETD	386
LES	全部	LAHF	全部
LSS	386	LODSB	全部
LFS	386	LODSW	全部
LGS	386	LODSD	386
DIV	全部	MOVSB	全部
MUL	全部	MOVSW	全部
IDIV	全部	MOVSD	386
IMUL	全部[3]	NOP	全部
NEG	全部	POPF	全部
NOT	全部	POPFD	386
ROL	全部	PUSHF	全部
ROR	全部	PUSHFD	386

[1] 386の特殊レジスタ転送を採用
[2] 8ビットの即値xを後に続けることができる
[3] 286の拡張命令

表2.1 Turbo Assemblerの命令セット（続き）

命令	プロセッサ	命令	プロセッサ
SAHF	全部	REPE	全部
SCASB	全部	REPZ	全部
SCASW	全部	REPNE	全部
SCASD	386	REPNZ	全部
STC	全部	SEGCS	全部[4]
STD	全部	SEGDS	全部[5]
STI	全部	SEGSS	全部[6]
STOSB	全部	EGES	全部[7]
STOSW	全部	SEGFS	386[8]
STOSD	386	SEGGS	386[9]
WAIT	全部	RET	全部
XLATB	全部	RETN	全部[10]
STOS	全部	RETF	全部[11]
SCAS	全部	JA	全部[12]
LODS	全部	JNBE	全部[12]
XLAT	全部	JAE	全部[12]
MOVS	全部	JNB	全部[12]
CMPS	全部	JNC	全部[12]
PUSHA	186-386	JB	全部[12]
PUSHAD	386	JNAE	全部[12]
POPA	186-386	JC	全部[12]
POPAD	386	JBE	全部[12]
LEAVE	186-386	JNA	全部[12]
INSB	186-386	JE	全部[12]
INSW	186-386	JZ	全部[12]
INSD	386	JG	全部[12]
OUTSB	186-386	JNLE	全部[12]
OUTSW	186-386	JGE	全部[12]
OUTSD	386	JNL	全部[12]
INS	186-386	JL	全部[12]
OUTS	186-386	JNGE	全部[12]
BOUND	186-386	JLE	全部[12]
ENTER	186-386	JNG	全部[12]
LOCK	全部	JNE	全部[12]

[4] CSオーバライドを使用（後に命令を続けることができる）
[5] DSオーバライドを使用（〃）
[6] SSオーバライドを使用（〃）
[7] ESオーバライドを使用（〃）
[8] FSオーバライドを使用（〃）
[9] GSオーバライドを使用（〃）
[10] 明示的nearリターン
[11] 明示的farリターン
[12] JUMPS実行時にnear/far引数を指定可能

表2.1 Turbo Assemblerの命令セット（続き）

命令	プロセッサ	命令	プロセッサ
REP	全部	JNZ	全部[12]
JNO	全部[12]	SETA	386
JNS	全部[12]	SETAE	386
JNP	全部[12]	SETB	386
JPO	全部[12]	SETBE	386
JO	全部[12]	SETC	386
JP	全部[12]	SETE	386
JPE	全部[12]	SETG	386
JS	全部[12]	SETGE	386
LOOP	全部[13]	SETL	386
LOOPE	全部[14]	SETLE	386
LOOPZ	全部[14]	SETNA	386
LOOPNE	全部[14]	SETNAE	386
LOOPNZ	全部[14]	SETNB	386
LOOPW	全部[15]	SETNBE	386
LOOPWE	全部[16]	SETNC	386
LOOPWZ	全部[16]	SETNE	386
LOOPWNE	全部[16]	SETNG	386
LOOPWNZ	全部[16]	SETNGE	386
LOOPD	386[17]	SETNL	386
LOOPDE	386[17]	SETNLE	386
LOOPDZ	386[17]	SETNO	386
LOOPDNE	386[17]	SETNP	386
LOOPDNZ	386[17]	SETNS	386
JCXZ	全部[18]	SETNZ	386
JECXZ	386[19]	SETO	386
BT	386	SETP	386
BTC	386	SETPE	386
BTR	386	SETS	386
BTS	386	SETZ	386
BSF	386		
BSR	386	ARPL	286p+386p
MOVSX	386	CLTS	286p+386p
MOVZX	386	LLDT	286p+386p
SHLD	386	LMSW	286p+386p
SHRD	386	LTR	286p+386p

[12] JUMPS実行時にnear/far引数を指定可能
[13] セグメントサイズで決まるループのオペランドサイズ
[14] LOOPと同様
[15] ループのオペランドサイズは常にワード（CX）
[16] LOOPWと同様
[17] ループのオペランドサイズは常にダブルワード（ECX）
[18] JCXZのオペランドサイズは常にワード（CX）
[19] JECXZのオペランドサイズは常にダブルワード（ECX）

表2.1 Turbo Assemblerの命令セット（続き）

命令	プロセッサ	命令	プロセッサ
SLDT	286p+386p	FXAM	8087-387
SMSW	286p+386p	FXTRACT	8087-387
STR	286p+386p	FXAM	8087-387
VERR	286p+386p	FXTRACT	8087-387
VERW	286p+386p	FYL2X	8087-387
LGDT	286p+386p	FYL2XP1	8087-387
LIDT	286p+386p	FNCLEX	8087-387
SGDT	286p+386p	FNDISI	8087-387
SIDT	286p+386p	FNENI	8087-387
LAR	286p+386p	FNINIT	8087-387
LSL	286p+386p	FADDP	8087-387
FMULP	8087-387	FDIVP	8087-387
FADD	8087-387	FDIVRP	8087-387
FDIV	8087-387	FSUBP	8087-387
FDIVR	8087-387	FSUBRP	8087-387
FMUL	8087-387	FXCH	8087-387
FSUB	8087-387	FCOMPP	8087-387
FSUBR	8087-387	FFREE	8087-387
FCOM	8087-387	FIADD	8087-387
FCOMP	8087-387	FICOM	8087-387
FST	8087-387	FICOMP	8087-387
F2XM1	8087-387	FIDIV	8087-387
FABS	8087-387	FIDIVR	8087-387
FCHS	8087-387	FIMUL	8087-387
FCLEX	8087-387	FIST	8087-387
FDECSTP	8087-387	FISUB	8087-387
FDISI	8087-387	FISUBR	8087-387
FENI	8087-387	FLDCW	8087-387
FINCSTP	8087-387	FSTCW	8087-387
FINIT	8087-387	FSTSW	8087-387
FLD1	8087-387	FNSTCW	8087-387
FLDL2E	8087-387	FNSTSW	8087-387
FLDL2T	8087-387	FLDENV	8087-387
FLDLG2	8087-387	FRSTOR	8087-387
FLDLN2	8087-387	FSAVE	8087-387
FLDPI	8087-387	FSTENV	8087-387
FLDZ	8087-387	FNSAVE	8087-387
FNOP	8087-387	FNSTEVN	8087-387
FPATAN	8087-387	FLD	8087-387
FPREM	8087-387	FSTP	8087-387
FPTAN	8087-387	FILD	8087-387
FRNDINT	8087-387	FISTP	8087-387
FSCALE	8087-387	FBLD	8087-387
FSQRT	8087-387	FBSTP	8087-387
FIST	8087-387	FWAIT	8087-387
FSETPM	287-387	FCOS	387
FPREM1	387	FPREMI	387
FSIN	387	FSINCOS	387
FUCOM	387	FUCOMP	387
FUCOMPP	387		

NEC PC-9801シリーズおよびIBM PC/XT

これまでは8086のアセンブリ言語に焦点をあてて説明してきました。重要なことは，8086プロセッサはコンピュータシステムのほんの一部に過ぎないことと，コンピュータのハードウェアや使用するオペレーティングシステムによってアセンブリ言語のプログラミング手法は異なるということです。

8086プロセッサ用に書かれたソフトウェアの大部分（たぶん現在まで作られたコンピュータソフトウェアの大部分）は，MS-DOSを搭載したNEC PC-9801シリーズやIBM PC, XT，およびそれらの互換機（以後IBM　PCと総称します）用のプログラムです。Turbo Assembler自身もその例外ではなく，MS-DOSのもとで走る上記の機種用のプログラムです。あなたもこのTurbo AssemblerをNEC PC-9801やIBM PCでお使いになること思います。

たとえハードウェアやオペレーティングシステムの知識が無くとも，アセンブラプログラムは走ります。しかしこれらの知識がないと入出力処理を行なうこともできなければ，プログラム自身を終了させることもできません。このマニュアルでNEC PC-9801シリーズやIBM PCの性能やソフトウェアのすべてを説明している余裕はありませんが，それらの基本的な性能については触れておきます。この章の最初で紹介した書籍を読むことをお勧めします。

入出力デバイス

NEC PC-9801シリーズやIBM PCの入出力デバイスには，キーボード，ディスプレイ，モニタ，フロッピィディスクドライブなどがあります。他にもモデム，プリンタ，マウス，ハードディスクなどがよく使われるデバイスです。これらのデバイスはI/Oポートまたはメモリ（またはその両方）を介して，非常に複雑な一連のアクセス処理によってコントロールされています。たとえばIBM PCのCGA（カラーグラフィックアダプタ）の新ビデオモードを使う場合は，30の**OUT**命令が必要です。キーボード，モデム，ディスクをコントロールする場合はもっと複雑になります。

だからといって，複雑なコントロールシーケンスをマスターしなければアセンブラプログラムを作れない，というわけではありません。システムソフトウェア（DOS）がこれらの装置の面倒を見てくれます。

システムソフトウェア

システムソフトウェアは，アプリーションソフトウェア（Turbo AssemblerやQuattroなど）とコンピュータハードウェアの間にあってコントロールおよびインターフェース機能を司るソフトウェアです（図2.10を参照してください）。

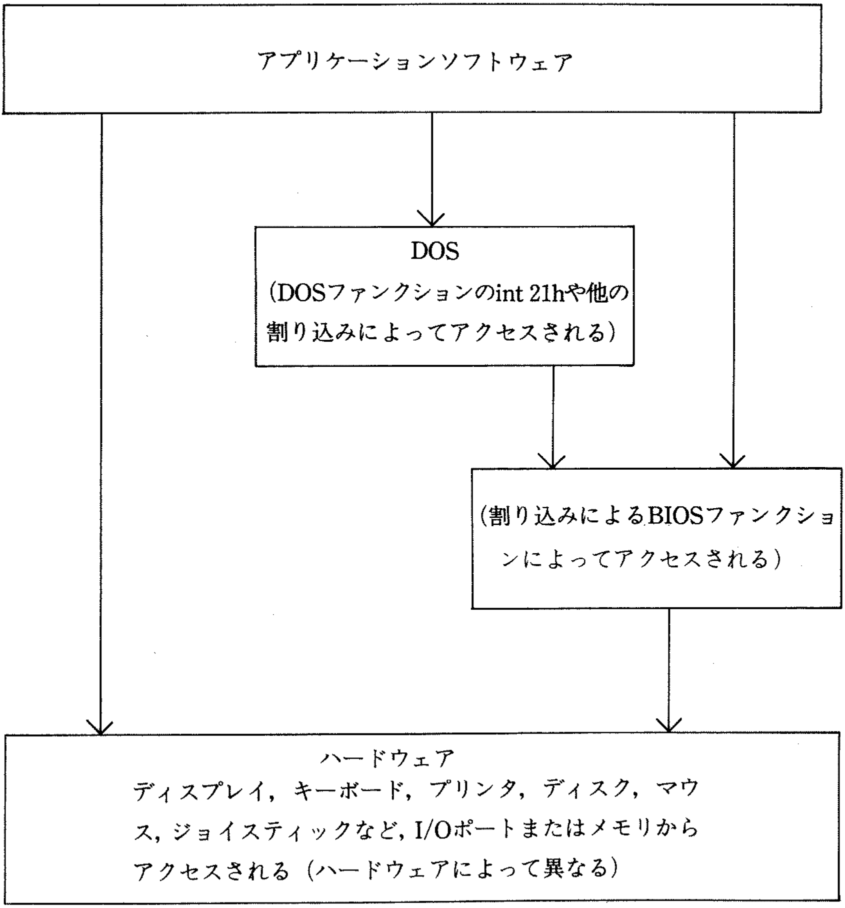


図2.10　コントロールとインターフェース機能をもつDOSとBIOSのシステムソフトウェア

特にシステムソフトウェアは個々のデバイスに対する複雑なハンドラとしての機能をも

っています。たとえばコンピュータに1回のキー押下を処理させるためには数行のアセンブラ命令が必要となります。しかしプログラマはそのプログラムを書く必要はなく，システムファンクションを1回呼び出すだけでこと足ります。これはDOSとBIOS（Basic Input/Output System）という2つのシステムソフトウェアのおかげです。

図2.10からもわかるように，DOSとBIOSのシステムソフトウェアはアプリケーションソフトウェアとコンピュータハードウェアの間にあるコントロールとインターフェースのプログラムです。アプリケーションソフトウェアはハードウェアを直接コントロールすることも可能です。しかしいかなる場合も，DOSファンクションまたはBIOSファンクションを使うべきです。

DOS

DOS（Disk Operating Systemの略，MS-DOSやPC-DOSなど）は，コンピュータの電源を入れた直後のディスクの読み込みから電源を切るまでの間コンピュータをコントロールするプログラムです。DOSは貴重な640Kバイトのメインメモリの一部を占有します。DOSなしではコンピュータは ただの箱 になってしまいます。画面上にプロンプト（A>またはC>などと表示されています）を表示したり，DIRなどのコマンドを受け付けて実行するのもすべてDOSの仕事です。

上記の処理は私たちの目に見える範囲のことです。DOSは実にさまざまな仕事をしており，アプリケーションソフトウェアは頻繁にその機能を利用しています。アプリケーションソフトウェアがディスクの読み書き，キー入力の検出，メモリのアクセス，他のプログラムの起動，時刻の読み込みやセットなどを行なう際は必らずDOSを経由します。たとえば次のプログラムは，画面出力のDOSファンクションを使って画面上のカーソル位置に文字Aを表示します。

```
.
.
.
mov   ah,2          ; 画面に1文字を表示するDOSファンクション
mov   dl,'A'        ; 'A'は表示する文字
int   21h           ; DOSを呼び出してファンクションを実行
.
.
.
```

プログラムでキーボードやファイルの読み書きまたはプリンタへの印字などを行なう場合は，DOSファンクションを必らず使ってください。DOS自身もアセンブラプログラムですから，DOSの仕事をユーザのプログラムにやらせることはもちろん可能です。しかしそ

うするのは賢明な方法とはいえません。NEC PC-9801やIBM PCの互換機のハードウェアはメーカや機種によって異なっています。DOSはこれらのハードウェアの違いをうまく吸収してくれるのです。DOSを使わないで直接ハードウェアをコントロールするプログラムは，他機種のコンピュータでは動かない可能性があります。

また,DOSを使わないプログラムは他のプログラム(SideKickやSuperKeyなどのメモリ常駐のプログラム）といっしょに使うことはむずかしくなります。DOSにはアプリケーションソフトウェアが必要とする機能があらかじめ用意されていますので，あらためてハードウェアの制御プログラムを作る必要はありません。DOSファンクションを利用することです。

DOSファンクションについておすすめできる入門書として，IBMの「***DOSリファレンスマニュアル***」があります。参照してください。

DOSファンクションに必要なものがない場合は，BIOSファンクションを使ってください。BIOSファンクションについては後で簡単に説明します。さしあたって入力，出力，プログラムの終了に使う基本的なDOSファンクションを見てみましょう。

キーの読み込み

キーボードからのキー入力はコンピュータと人間のもっとも基本的な会話手段です。アセンブラプログラムがDOSを利用してキーの読み込みを行なうには多くの方法があります。その一例を次に示します。

キーを読み込むもっとも単純な方法はキーボード読み込みファンクション（DOSファンクション 番号1)を使うことです。DOSファンクションを使うためにはAHレジスタにファンクション番号をセットして**INT 21h**命令を実行します（**INT**命令の働きは少し複雑です。ここではDOSファンクションを呼び出すためには，**INT 21h**命令を実行することを憶えておいてください)。以後キー入力された文字コードがALレジスタにセットされます。

次のプログラムを実行するとキーボードから入力された文字コードがALレジスタにセットされます。

```
.
.
.
mov   ah,1
int   21h
.
.
.
```

キーが押されない場合はDOSはいずれかのキーが押されるまで待ちます。したがってこの関数は非常に長時間呼び出されたままになります。

画面への文字表示

キー入力が人間のソフトウェアとの会話手段だとすると，画面表示はその逆にあたります。現在のパソコンにはカラー表示から高精度表示まで，画面に関するすべての制御機能が備っています。はじめに文字の表示について説明します。

DOSのファンクション番号2は文字を画面に表示するためのファンクションです。AHレジスタに2を，DLレジスタに文字をセットし，DOSファンクションの**INT 21h**をコールします。次にプログラム例を示します。

```
.
.
.
mov  ah,1
int  21h              ; キーの検出
mov  ah,2
mov  dl,al            ; 文字をALからDLへ移動
int  21h              ; 文字を表示
.
.
.
```

文字や文字列の読み込みや表示を行なうファンクションはこの他にもあり，このマニュアルのプログラム例でも使われています。ここではDOSファンクションのすべてを説明しませんが，巻末の参考文献を読んでDOSファンクションについて勉強することをぜひお勧めします。DOSファンクションはアセンブラプログラミングのキーポイントになります。

アセンブリ言語を使ったキーボード，画面，ファイルの入出力についてのポイントがもう1つあります。C言語の**scanf**や**prinf**，Pascalの***Readln***や***Writeln***を使ったことのある方は，DOS（またはアセンブリ言語）がフォーマット化された入力や出力をサポートしていないと聞いて驚くかもしれません。DOSは文字や文字列の入出力処理を行なうだけです。C言語では整数変数 *i* をプリントするために必要な処理は次の*1*行です。

```
printf("%d¥n",i);
```

C言語では整数の値（16ビットのメモリアドレスにストアされます）は自動的にASCII文字列に変換され，その文字が表示されます。一方アセンブラで文字を表示させる場合は，プログラムで変数の値を文字列に変換しなければなりません。同様に，DOSもキーボードから文字または文字列を読み込むだけなので，やはりアセンブラプログラムでデータ型を変換する必要があります。

次の章の終わりでアセンブリ言語を使って変数の値を表示するために必要なことをすべて説明します。ここではDOSファンクションを使えば文字や文字列を表示できることを憶

えておいてください。データをDOSが扱える型から文字型へ変換するのはプログラマの仕事です。

プログラムの終了

プログラムの読み書きを少し覚えましたので，1行分のキー入力を画面にエコー(入力した文字を画面上に反映させること）する簡単なプログラムを作ってみましょう。使用するDOSファンクションは3種類で，うち2種類はいま見てきたものです。残りの1つはプログラムを終了するファンクションです。

CまたはPascalに慣れているプログラマはアセンブラプログラムでもメインプログラムの実行が終わればプログラムから抜けると思いがちですが，そうではありません。アセンブラプログラムを終了させる場合もDOSファンクションをコールしなければなりません。

プログラム終了のDOSファンクションにはいくつかありますが，DOSファンクション番号4Chをコールすることをお勧めします（10進数に慣れている方は76を指定してください)。〝エコー〟プログラムは次のようになります。

```
    DOSSEG
    .STACK 100h
    .MODEL SMALL
    .CODE
EchoLoop:
    mov   ah,1          ; キー読み込みのDOSファンクション番号
    int   21h           ; キー読み込み
    cmp   al,13         ; リターンキーか?
    jz    EchoDone      ; Yes - EchoDoneへジャンプ
    mov   dl,al         ; DLへ文字を移動
    mov   ah,2          ; 文字表示のDOSファンクション番号
    int   21h           ; 文字表示
    jmp   EchoLoop      ; 次の文字の処理へ
EchoDone:
    mov   ah,4ch        ; プログラム終了のDOSファンクション番号
    int   21h           ; プログラム終了
    END
```

プログラムを正しく入力して実行してください。キーを押すと文字が2回表示されたでしょう。1つ目の文字はDOSがエコーしたもので，2つ目の文字はプログラムがエコーしたものです。このプログラム例で重要なことは，キーの読み込み，画面への表示，プログラムの終了のすべての処理をDOSファンクションで実行していることです。

BIOS

DOSファンクションといえどもプログラマの要求に応えられないことがあります。そん

なときはNEC PC-9801またはIBM PCのBIOS（Basic Input/Output System）の出番です。DOSやアプリケーションソフトウェアと異なり，BIOSはディスクからロードされるわけでも640Kバイトの汎用メモリを使うわけでもありません。BIOSはROM（Read Only Memory）にストアされています。システムファンクションを含むROMは，8086の予約されたアドレス空間に存在します。

BIOSはコンピュータのソフトウェアの中でもっともレベルの低いものです。DOSでさえもBIOSファンクションを使ってハードウェアをコントロールしています。ハードウェアを直接コントロールするよりもBIOSを使うことをお勧めします。なぜなら，DOSと同様にBIOSはさまざまなコンピュータやデバイスの違いをを吸収してくれるからです。一方BIOSを使うプログラムは他のプログラムと衝突することがあり，コンピュータの機種によっては移植性が悪くなります。したがってBIOSファンクションよりもむしろDOSファンクションを使うべきです。

ディスプレイモードの選択（IBM PCのとき）

DOSはコンピュータの豊富な画面表示機能をサポートしていません。BIOSを使う最大の理由は画面制御のためです。BIOSファンクションをコールすれば画面モード，カラーコントロール，ディスプレイアダプタ情報などを設定することができます。次のプログラムはBIOSファンクションをコールして画面をCGA用の4色グラフィックモードに設定します。

```
  .
  .
  .
mov  ah,0              ; BIOSのセットモードファンクション
mov  al,4              ; 320x200,4色グラフィックスモード番号
int  10h               ; BIOSビデオ割り込みを実行してモードを設定
  .
  .
  .
```

前にも述べたように，ビデオモードを設定するためには**OUT**命令を30回以上実行する必要があります。したがってBIOSのセットモードファンクションを使えば大幅に手間が省けます。BIOSには画面制御を除いてはキーボード読み込みやディスクコントロールの多様なファンクションが備っています。ただし，一般的にはこれらの処理はDOSファンクションで実行しないほうがよいでしょう。

BIOSファンクションについてはIBMの「*BIOS Inteface Technical Reference*」が入門書として最適です。

場合によってはハードウェアを直接制御することも必要

DOSファンクションまたはBIOSファンクションを使う理由は以上です。場合によってはハードウェアを直接制御することも必要となります。DOSやBIOSではシリアルポートを充分サポートしていないため，たとえば通信ソフトは**IN**命令や**OUT**命令を使ってシリアルポートを直接コントロールしなければなりません。同様にDOSはグラフィックスをサポートせず，BIOSはサポートしても速度が遅いため，高性能グラフィックスを使うには直接ディスプレイメモリにアクセスしなければなりません。

ハードウェアの直接制御はやむをえない場合にのみ行なうのが基本ルールです。つまり，DOSファンクションやBIOSファンクションがある場合はできる限りそれを使うようにし，ない場合にのみハードウェアを直接コントロールしてください。プログラミングの目的はあくまでも役に立つプログラムを作ることであり，必らずしもこのルールを厳守する必要はありません。しかしこの基本ルールを守れば問題は少なくて済むはずです。

他の資源

アセンブリ言語のプログラマにとっては，ハードウェアおよびソフトウェアの資源は豊富にあります。ここでは資源の説明はしませんが，いくつかをリストアップしておきます。詳細についてはこの章のはじめを参照してください。

■ANSI.SYSドライバはBIOSファンクションを使うことなく高度な画面制御を行ないます（IBM-PCのとき）

■システムタイマは日付/時刻クロックをサポートし，スピーカと高精度タイミング信号によって音声発生機能もサポートします（IBM-PCのとき）

■オプションの8087数値演算コプロセッサを使えば，浮動小数点の演算速度が大幅に向上します。

第3章

コマンドラインリファレンス

この章はTurbo Assemblerのコマンドラインオプションに慣れていただくことを目的としています。アセンブラの動作を指定するために使う各コマンドラインオプションの使い方について解説します。またコンフィギュレーションファイルについても説明します。

DOSからTurbo Assemblerを開始する

Turbo Assemblerはパワフルでフレキシブルなコマンドラインシンタックスを備えています。次のように引数を付けないでTurbo Assemblerを起動すると，コマンドラインオプションについてのヘルプメッセージとアセンブルするファイルの指定方法が画面に表示されます。

```
TASM
```

図3.1にその画面を示します。

```
Turbo Assembler  Version 1.0  Copyright (c) 1988 by Borland International

Syntax:  TASM [options] source [,object] [,listing] [,xref]

/a,/s          Alphabetic or Source-code segment ordering
/c             Generate cross-reference in listing
/dSYM[=VAL]    Define symbol SYM = 0, or = value VAL
/e,/r          Emulated or Real floating-point instructions
/h,/?          Display this help screen
/iPATH         Search PATH for include files
/jCMD          Jam in an assembler directive CMD (eg. /jIDEAL)
/kh#,/ks#      Hash table capacity #, String space capacity #
/l,/la         Generate listing: l=normal listing, la=expanded listing
/ml,/mx,/mu    Case sensitivity on symbols: ml=all, mx=globals, mu=none
/n             Suppress symbol tables in listing
/p             Check for code segment overrides in protected mode
/t             Suppress messages if successful assembly
/w0,/w1,/w2    Set warning level: w0=none, w1=w2=warnings on
/w-xxx,/w+xxx Disable (-) or enable (+) warning xxx
/x             Include false conditionals in listing
/z             Display source line with error message
/zi,/zd        Debug info: zi=full, zd=line numbers only
```

図3.1　Turbo Asemblerコマンドライン

コマンドラインオプションを使えばアセンブルしたいファイル名をいくつでも指定できます。またその際のアセンブル方法を指定することもできます。

コマンドラインの一般形式は次のとおりです。

```
TASM fileset [; fileset]...
```

セミコロン（；）でファイル名を区切って複数ファイルを並べれば，それらのファイルのアセンブルを指定できます。必要なら次のようにファイルごとにオプションを指定する

こともできます。

```
TASM /E FILE1; /A FILE2
```

この例では/**E**オプションでファイルFILE1.ASMをアセンブルし，/**A**オプションでファイルFILE2.ASMをアセンブルします。

コマンドラインの一般形式における*fileset*は次のようになります。

```
[option]... sourcefile [[+] sourcefile]...
            [,[objfile] [, [listfile], [, [xreffile]]]]
```

この文法によれば，アセンブルしたいファイル名の前に任意のオプションを付けることができます。複数ファイルのアセンブルを指定するときは通常のファイル名の他，DOS形式のワイルドカードの＊や？が使えます。拡張子（エクステンション）を省略すると自動的に.ASMが付加されます。たとえばカレントディレクトリのすべての.ASMファイルをアセンブルする場合は，次のようにコマンドを入力します。

```
TASM *
```

複数のファイルをアセンブルしたい場合は，次のようにプラス記号（＋）またはスペースでファイル名を区切ってください。

```
TASM MYFILE1 + MYFILE2
```

または，

```
TASM MYFILE1 MYFILE2
```

アセンブルしたいファイル名の後に任意のオブジェクトファイル名，リストファイル名，クロスリファレンスファイル名を指定できます。オブジェクトファイル名またはリストファイル名を省略した場合はソースファイルと同一名で拡張子が.OBJのオブジェクトファイルが自動生成されます。

リストファイルは指定しない限り作られません。リストファイルを指定するにはオブジェクトファイル名のあとにカンマ(,)を付けてリストファイル名を指定してください。リストファイル名を省略した場合はソースファイルと同一名で拡張子が.LSTのリストファイルが自動生成されます。リストファイル名の拡張子を省略すれば.LSTが付けられます。

クロスリファレンスファイルも指定しない限り作られません。クロスリファレンスファイルを指定するにはオブジェクトファイル名のあとにカンマ(,)を付けてそのファイル名を指定してください。クロスリファレンスファイル名を省略した場合は，ソースファイルと同一名で拡張子が.XRFのファイルが自動的に生成されます。クロスリファレンスファイ

ル名の拡張子を省略すれば.LSTが付けられます。グローバルクロスリファレンスユーティリティ（TCREF）によってクロスリファレンスファイルを処理する方法については付録Dを参照してください。

デフォルトのオブジェクトファイル名を使い，しかもリストファイルを作りたい場合は次のようにオブジェクトファイル名とリストファイル名をカンマ（,）で区切ってください。

```
TASM FILE1,,TEST
```

この例では，ソースファイルFILE1.ASMがアセンブルされ，オブジェクトファイルFILE1.OBJとリストファイルTEST.LSTを作られます。

デフォルトのオブジェクトファイル名とリストファイル名を使い，しかもクロスリファレンスファイルを作りたい場合はファイル名をカンマ（,）で区切ってください。

```
TASM MYFILE,,,MYXREF
```

ワイルドカードを使ってアセンブルしたいソースファイルを指定した場合は，オブジェクトファイル名やリストファイル名もそのワイルドカードによって指定することができます。カレントディレクトリにXX1.ASMとXX2.ASMという2つのファイルがある場合，次のコマンドラインを入力すると，

```
TASM XX*,YY*
```

*XX*で始まるすべてのファイルをアセンブルし，*YY*で始まるオブジェクトファイルを作り，ファイル名の残りの部分はソースファイル名から流用します。したがってオブジェクトファイル名はYY1.OBJ, YY2.OBJとなります。

オブジェクトファイルは不要でリストファイルが必要な場合，またはリストファイルが不要でクロスリファレンスファイルが必要な場合は，ヌルデバイス（NUL）をファイル名として指定することができます。次の例はソースファイルFILE1.ASMをアセンブルしてオブジェクトファイルFILE1.OBJとクロスリファレンスファイルFILE1.XRFを作ります。リストファイルは作りません。

```
TASM FILE1,,NUL,
```

コマンドラインのオプション

コマンドラインのオプションを指定すればアセンブラの動作，画面表示，リスト形式，オブジェクト形式などをコントロールできます。Turbo Assemblerには，次のような何もしないオプションもあります。これらはMASMの現行バージョンまたは以前のバージョンと互換性を保つために必要です。

/B　　バッファサイズの設定
/V　　拡張情報の表示

オプションは大文字・小文字あるいはそれらの組み合わせで入力できます。また/**I**または/**J**を複数指定する場合を除いて，オプションの入力順は自由です(/**I**, /**J**は順番に処理されます)。/**D**オプションを使うときは事前にシンボル定義を行なってください。

注意：ソースコードの中に疑似命令を入れておくと，コマンドラインオプションより優先順位が高くなります。

この章のはじめの図3.1にTurbo Assemblerのコマンドラインオプションの一覧を示してあります。次のページから各オプションについて説明します。

/A

機能　アルファベットセグメントオーダを指定します。

形式　/A

説明　**/A**オプションはオブジェクトファイル内のセグメントの順番をアルファベット順に指定します。ソースコード内に**.ALPHA**疑似命令を入れておいても同じです。このオプションは通常はIBMまたはマイクロソフトアセンブラの初期バージョン用に書かれたソースファイルをアセンブルするときにのみ使います。
/Sオプションは/**A**オプションの働きを取り消し，セグメントの順番をデフォルト（シーケンシャルセグメントオーダ）に戻します。
ソースコード内に**.SEQ**疑似命令を指定してシーケンシャルセグメントオーダを指定すると，コマンドラインの/**A**オプションに優先します。

例

```
TASM /A TEST1
```

セグメントの順番がアルファベット順のオブジェクトファイルTEST1.OBJを作ります。

/B

形式　/B

説明　/**B**オプションはMASMとの互換性を保つために使います。アセンブラに対してはダミーとなり，何の動作もしません。

/C

機能　リストファイル内にクロスリファレンス情報を付加します。

形式　`/C`

説明　/Cオプションはリストファイル内にクロスリファレンス情報を付加します。Turbo Assemblerはリストファイルの最後にあるシンボルテーブルにクロスリファレンス情報を付加します。クロスリファレンス情報を見るためには，コマンドライン上で/Lオプションを指定してリストファイルを指定します。
クロスリファレンスは各シンボルが定義されている行とそれを参照しているすべての行を表示します。

例

```
TASM /L /C TEST1
```

シンボルテーブル内にクロスリファレンス情報を持つリストファイルを作ります。

/D

機能　シンボルを定義します。

形式　`/Dsymbol[=value または =expression]`

説明　**/D**オプションはソースファイルのためのシンボルを定義します。これによってソースコードの先頭行で=疑似命令を使って定義することと同じになります。このオプションは，コマンドライン上で必要なだけ何度でも使うことができます。
シンボルは他のシンボルと等しくしたり定数として定義することができます。等号(=)の右側に演算子を伴う式を定義することはできません。たとえば/DX=9や/DX=Yは定義できますが/DX=Y-4は定義できません。

例

```
TASM /DMAX=10 /DMIN=2 TEST1
```

このコマンドラインは**MAX**と**MIN**の2つのシンボルを定義します。ソースファイルTEST1.ASM中の他の文はこれらのシンボルを参照することができます。

/E

機能　浮動小数点のエミュレータ命令を生成します。

形式　/E

説明　**/E**オプションは浮動小数点エミュレータ命令を生成するように指定します。この命令はソフトウェアの浮動小数点エミュレータによって実行されます。このオプションは，プログラム中に80x87数値演算コプロセッサの機能をシミュレートする浮動小数点エミュレーションライブラリがある場合に使います。

通常このオプションは，アセンブラモジュールが高級言語（ただし浮動小数点エミュレータを使う高級言語）で書かれたプログラムの一部であるときにのみ使います（Turbo C, Turbo Pascal, Turbo Basic, Turbo Prologはすべて浮動小数点エミュレータをサポートしています)。ライブラリはコンパイラのスタートアップコードによって初期化されている可能性が大きいため，アセンブラプログラムをエミュレーションライブラリとリンクすることはできません。

/Rオプションは**/E**オプションと逆の働きをします。すなわち数値演算コプロセッサによってのみ実行される浮動小数点の実命令のアセンブルを指定します。

ソースファイル中の**NOEMUL**疑似命令は，コマンドラインの**/E**オプションに優先します。

コマンドラインの**/E**オプションは，ソースファイルの先頭で**EMUL**疑似命令を使うのと同じ働きをします。またコマンドラインの**/JEMUL**と同じ働きをします。

例

```
TASM /E SECANT
TCC -f TRIG.C SECANT.OBJ
```

第1行目は疑似浮動小数点命令を使ってモジュールをアセンブルします。第2行目は浮動小数点エミュレーションを使ってC言語のソースモジュールをコンパイルし，それをオブジェクトファイルとリンクします。

/Hまたは/?

機能　ヘルプ画面を表示します。

形式　`/H または /?`

説明　/Hオプションはコマンドラインの形式を説明するヘルプ画面を表示することを指定します。ヘルプ画面にはオプション一覧と指定可能なさまざまなファイル名が示されます。/?オプションも同じ働きをします。

例　`TASM /H`

/I

機能　インクルードファイルのパスを設定します。

形式　`/Ipath`

説明　/Iオプションはソースファイル内の**INCLUDE**疑似命令に対するファイルのパス（*path*）を指定します。コマンドラインには複数の/Iオプションを指定することができます（指定できる数はRAMによってのみ制限されます)。

Turbo Assemblerは**INCLUDE**疑似命令を見つけると，指定されているディレクトリパスまたは単なるファイル名に基づいて**インクルードファイル**の場所を認識します。

ファイル名の一部としてディレクトリパスを指定すると，Turbo Assemblerは最初にそのパスを検索し，次にコマンドラインの/Iオプションで指定されたディレクトリを指定順に検索し，最後にコンフィギュレーションファイル内の/Iオプションで指定されたディレクトリを検索します。

ファイル名の一部としてディレクトリパスを指定しない場合は，Turbo Assemblerは最初にコマンドラインの/Iオプションで指定されたディレクトリを検索し，次にコンフィギュレーションファイル内の/Iオプションで指定されたディレクトリを検索し，最後にカレントディレクトリを検索します。

例

```
TASM /I¥INCLUDE /ID:¥INCLUDE TEST1
```

ソースファイル内に次の文があれば，

```
INCLUDE MYMACS.INC
```

Turbo Assemblerは最初に¥INCLUDE¥MYMACS.INCを，次にD:¥INCLUDE¥MYMACS.INCを検索します。それでもなおファイルが見つからない場合は，カレントディレクトリのMYMACS.INCを検索します。またソースファイル内に

```
INCLUDE INCS¥MYMACS.INC
```

という文があればTurbo Assemblerは最初に¥INCS¥MYMACS.INCを，次に

¥INCLUDE¥MYMACS.INCを，最後にD:¥INCLUDE¥MYMACS.INCを検索します。

/J

機能　アセンブラスタートアップ疑似命令を定義します。

形式

```
/Jdirective
```

説明　/Jオプションはソースファイルの先頭行の前にアセンブルする疑似命令を指定します。*directive*はTurbo Assemblerの任意の疑似命令で，引数を伴わないものなら何でもかまいません(たとえば**.286, IDEAL, %MACS, NOJUMPS**など)。Turbo Assemblerの疑似命令については**リファレンスガイド**を参照してください。
コマンドラインには複数の/Jオプションを指定できます。複数個指定した場合は左から順番に処理されます。

例

```
TASM /J.286 /JIDEAL TEST1
```

この例では80286命令とIdealモードの式解釈を可能にして，ファイルTEST1.ASMをアセンブルします。

/KH

機能　シンボルの最大値を設定します。

形式　`/KHnsymbols`

説明　**/KH**オプションはプログラムが持つことのできる最大シンボル数を設定します。最大シンボル数のデフォルトは8,192個ですが，このオプションを使えばシンボル数を*nsymbols*（最大32,768個）まで増やすことができます。

アセンブルの際に**Out hash space**のメッセージが表示された場合は，このオプションを使ってください。

またこのオプションによって最大シンボル数をデフォルト以下にすることもできます。プログラムのアセンブルに際してメモリが足りないときに，最大シンボル数を減らせば，メモリの一部が開放されます。

例

```
TASM /KH100000 BIGFILE
```

この例は，ファイルBIGFILEをアセンブルの際に10000個のシンボル用のメモリを確保することを指定します。

/KS

機能　Turbo Assemblerの文字列スペースの最大サイズを設定します。

形式　`/KSkbytes`

説明　通常は文字列のサイズは自動的に決定されるため指定する必要はありません。しかしプログラムのアセンブルの際に**Out of string space**のメッセージが表示された場合は，このオプションを使って文字列スペースのサイズを増やしてください。100からはじめて徐々に値を大きくしていき，正常にアセンブルが終了する値を探してください。*kbytes*の最大値は255です。

例

```
TASM /KS150 SFILE
```

この例では150Kバイトの文字列スペースを確保します。

/L

機能　リストファイルを作成します。

形式　/L

説明　**/L**オプションはリストファイルの作成を指定します。コマンドラインで指定しなかった場合でも有効です。リストファイルはソースファイルと同じファイル名を持ち，拡張子は.LSTとなります。

例

```
TASM /L TEST1
```

この例はリストファイルTEST1.LSTを作成します。

/LA

機能　リストファイル内の高級言語インターフェースコードを表示します。

形式　/LA

説明　**/LA**オプションはリストファイル内のすべてのコードを表示します。コードの中には高級言語とのインターフェースである**.MODEL**疑似命令の結果コードも含まれます。

例

```
TASM /LA FILE1
```

/ML

機能　シンボルの大文字小文字の区別をつけてシンボルを扱います。

形式　/ML

説明　**/ML**オプションは，すべてのシンボル名を大文字小文字の区別をつけて扱うことを指定します。通常は大文字と小文字は区別されません(たとえば*ABCxyz, abcxyz, ABCXYZ*はいずれも同一シンボルを参照します)。**/ML**オプションを指定した場合は，これらの3種類のシンボルは明確に区別されます。その場合でもアセンブラのキーワードは，大文字または小文字のどちらを使うこともできます。キーワードはアセンブラの組み込みシンボルであり，命令ニーモニック，疑似命令，演算子などの特別な意味を持っています。

例

```
TASM /ML TEST1
```

ファイルTEST1.ASM内には次の文が含まれていなければなりません。

```
abc DW 0
ABC DW 1              ; 異なるシンボル
    Mov Ax,[Bp]       ; 大文字小文字が混在できるシンボル
```

/MU

機能　シンボルを大文字に変換します。

形式　/MU

説明　**/MU**オプションは，シンボルの大文字小文字の区別をしないことを指定します。**/ML**オプションで指定しない限り，シンボルの小文字は自動的に大文字に変換されます。

例

```
TASM /MU TEST1
```

すべてのシンボルを大文字に変換します（デフォルトです）。

```
EXTERN myfunc:NEAR
CALL myfunc                 ; 大文字と小文字の区別なし
```

/MX

機能　パブリックシンボルと外部シンボルの大文字小文字の区別をつけます。

形式　/MX

説明　**/MX**オプションは，パブリックシンボルと外部シンボルの大文字小文字の区別をつけるように指定します（ソースファイル内の）他のシンボルは大文字で扱われます。

大文字小文字の区別がされたままコンパイル・アセンブルされた他のモジュール内のプログラム（Turbo Cでコンパイルされたプログラムなど）をコールする際は，このオプションを使う必要があります。

例

```
TASM /MX TEST1;
```

ただしTEST1.ASMには次のような行が含まれていなければなりません。

```
EXTERN Cfunc:NEAR
myproc PROC NEAR
call  Cfunc
.
.
.
```

/N

機能　リストファイル内のシンボルテーブルを無効にします。

形式　/N

説明　このオプションは，リストファイルの最後に通常のシンボルテーブルを付けないことを指定します。通常はシンボルテーブルリストがファイルの最後に作られ，すべてのシンボルとその型，および値を示します。

例

```
TASM /L /N TEST1
```

この例はシンボルの値ではなくコードのみを示すリストファイルを作成します。

/P

機能　プロテクトモードでの不純コードをチェックします。

形式　`/P`

説明　**/P**オプションは，プロテクトモードにおいて不純コードを発生する命令に関する警告メッセージを出すことを指定します。**CS**オーバライドを使ってメモリにデータを転送する命令は不純とみなされます。なぜなら，これらの命令は特別な処置を施さない限り正しく働かないからです。
このオプションは80286や80386のプロテクトモードで動くプログラムを書く場合にのみ指定します。

例

```
TASM /P TEST1
```

ファイルTEST1.ASM内に次の文が含まれていると警告メッセージが出力されます。

```
     .286P
CODE SEGMENT
temp DW  ?
     mov CS:temp,0      ; プロテクトモードでは‘不純’
```

/R

機能　浮動小数点命令の実命令を生成します。

形式　/R

説明　**/R**オプションは（疑似浮動小数点命令のかわりに）浮動小数点命令の実命令を生成することを指定します。このオプションはプログラムを80x87数値演算コプロセッサを搭載したマシンで動かす場合に指定します。

/Eオプションは/**R**オプションの逆の働きをします。すなわち疑似浮動小数点命令を生成することを指定します。

ソースファイル内に**EMUL**疑似命令がある場合はコマンドラインの/**R**オプションに優先します。

/**R**コマンドラインオプションはソースファイルの先頭で**NOEMUL**疑似命令を使うのと同じ効果があります。また/**JNOEMUL**オプションも同様です。

例

```
TASM /R SECANT
TPC /$N+ /$E- TRIG.PAS
```

1行目は浮動小数点命令の実命令によってモジュールをアセンブルします。2行目は，アセンブラからのオブジェクトファイルをリンクする浮動小数点命令の実命令によって，Pascalのソースモジュールをコンパイルします。

/S

機能　シーケンシャルセグメントオーダを指定します。

形式　`/S`

説明　/Sオプションはオブジェクトファイル内にソースファイル内の順番と同じ順番でセグメントを置くことを指定します。/Aオプションをコンフィギュレーションファイル内に指定しない限りセグメントオーダが使われます。
ソースファイル内に.ALPHA疑似命令を入れてアルファベットセグメントオーダを指定した場合は，コマンドラインの/Sオプションに優先します。

例

```
TASM /S TEST1
```

この例ではソースファイル内に指定されたのと同じ順番のセグメントを持つオブジェクトファイルTEST1.OBJを作成します。

/T

機能　正常なアセンブル処理のメッセージ表示を取り止めます。

形式　/T

説明　/Tオプションはアセンブルに警告メッセージやエラーメッセージ以外の表示を取り止めます。

このオプションは，多くのモジュールをアセンブルする際に使えば，画面には警告メッセージとエラーメッセージだけが表示されるようになります。

例

```
TASM /T TEST1
```

/V

形式　/V

説明　/VオプションはMASMとの互換性を維持するために使います。アセンブルに対しては何の影響も与えません。

/W

機能　警告メッセージの表示をコントロールします。

形式

```
/W
W-[warnclass]
W+[warnclass]
```

説明　/Wオプションは表示する警告メッセージを選択します。

/Wのみを指定すると〝マイルド〟メッセージが有効となります。このメッセージ，はプログラムの効率を上げられる旨を示すだけです。

/Wに*warnclass*を付けないで指定すると，すべての警告メッセージが無効となります。/Wに*warnclass*を指定すると，その警告だけが無効となります。それぞれの警告メッセージは次のように3文字の識別子を持っています。

ASS	16ビットのセグメントを仮定します。
BRK	カッコが必要です。
ICG	コードを生成できません。
LCO	ロケーションカウンタがオーバフローしました。
OPI	IF条件がオープンです。
OPP	手続きがオープンです。
OPS	セグメントがオープンです。
OVF	算術結果がオーバフローしました。
PDC	パスに依存した構造です。
PRO	プロテクトモードでメモリに書き込むためにはCSオーバライドが必要です。
RES	ワードが確保されています。
TPI	Turbo Pascalの不正警告
PQK	[const]に定数を想定します。
ALN	セグメントアラインメント

*warnclass*を付けないで/W＋を指定すると，すべての警告メッセージが有効となります。上記の*warnclass*を付けた場合はその警告メッセージのみが有効となります。

デフォルトではinefficient code-generation（IGC）（効率の悪いコード生成）とwrite-to-memory in protected-mode（PRO）（プロテクトモードでのメモリ書き込み）以外のすべての警告メッセージを有効にしてアセンブルを開始します。ソースファイル中に**WARN**と**NOWARN**の疑似命令を入れると，特定の警告をソース行の特定の範囲内で使うかどうかをコントロールすることができます。詳細は**リファレンスガイド**の第3章を参照してください。

例

```
TASM /W TEST1
```

TEST1.ASM内の次の文によって警告メッセージが表示されます。**/W**オプションがなければ表示されません。

```
mov   bx,ABC        ; Inefficient code-generation警告
ABC = 1
```

TEST2.ASMに

```
TASM /W-OVF TEST2
```

が入っている場合は，次のコマンドラインを実行しても警告メッセージは表示されません。

```
dw  1000h * 20h
```

/X

機能　リストに偽条件を入れます。

形式　/X

説明　/**X**オプションを指定した場合は，条件**IF, IFNDEF, IFDEF**などが偽と判断されたときに条件ブロック内の文がリストファイルにも出力されます。またこのオプションを指定すると，通常はリストされない条件疑似命令自身がリストされます。リストファイルをコマンドラインまたは/**L**オプションで指定しなければなりません。さもないと/**X**オプションは無効となります。
.LFCOND, SFCOND, TFCOND疑似命令を使うと，/**X**オプションに優先します。

例　TASM /X TEST1

/Z

機能　エラーメッセージとともにソース行を表示します。

形式　/Z

説明　/**Z**オプションはエラーメッセージが発生するとソースファイルの対応行を表示することを指定します。エラーが発生した行はエラーメッセージの前に表示されます。このオプションを無効にするとエラーメッセージのみが表示されます。

例　TASM /Z TEST1

/ZD

機能　オブジェクトファイル内の行番号情報を使用可能にします。

形式　`/ZD`

説明　**/ZD**オプションはオブジェクトファイル内に行番号情報を入れることを指定します。ボーランド社のスタンドアロンデバッガ Turbo Debbugerは，ソースファイル内のカレントロケーションを表示します。ただしデバッガがデータ項目にアクセスするために必要な情報をオブジェクトファイルに格納しません。
Turbo Debbugerでプログラムデバッグ中にメモリが足りなくなった場合は，モジュールによって/**ZD**または/**ZI**を使いわけることができます。

例

```
TASM /ZD TEST1
```

/ZI

機能　オブジェクトファイル内のデバッグ情報を使用可能にします。

形式　`/ZI`

説明　/**ZI**オプションはすべてのデバッグ情報をオブジェクトファイルに付加することを指定します。この情報の中にはソースコード表示とデータ型情報の同期をとるための行番号レコードが含まれ，プログラムデータのチェックや変更に役立ちます。/**ZI**オプションを指定すれば，Turbo Debuggerのすべての機能を使うことが可能となり，プログラムのチェック，データ項目のチェックや変更に役立ちます。このオプションは，すべてのプログラムモジュールにもデバッグしたい特定モジュールに対しても使うことができます。また，オブジェクトプログラムと実行可能プログラムに情報の追加処理を施します。このためTurbo Debuggerを使ってプログラムを実行中にメモリが足りなくなった場合は，すべてのモジュールにはこのオプションを使わない方がよいでしょう。

例

```
TASM /ZI TEST1
```

間接コマンドファイル

コマンドラインの入力の際は，次のように任意の場所で単価記号（@）に続けて間接コマンドを指定することができます。

```
TASM /DTESTMODE @MYPROJ.TA
```

この例ではファイルMYPROJ.TAの内容がそのままの形でコマンドラインの一部になります。　頻繁に使うコマンドラインやファイルリストを別々のファイルに格納しておけば，コマンド全部をタイプする手間が省けます。コマンドライン上では複数の間接ファイルを指定したり，間接コマンドを通常の引数と混在して指定することができます。次に例を示します。

```
TASM @MYFILES @IOLIBS /DBUF=1024
```

このように標準ファイルやオプションの長いリストをファイルに格納しておけば，アセンブラの動作をその場でしかも簡単に変更することができます。

コマンドファイル中には，すべてのファイル名やオプションを1行で入れることも，複数行にわけておくこともできます。

コンフィギュレーションファイル

Turbo Assemblerでは，カレントディレクトリのコンフィギュレーションファイル内に頻繁に使用するオプションを入れておくことができます。これによりTurbo Assemblerを立ち上げた際にカレントディレクトリ内のTASM.CFGというファイルの有無がチェックされます。DOS 3.xまたはそれ以上のバージョンで実行している場合は，Turbo Assemblerがロードされたディレクトリがチェックされます。ファイルが見つかった場合はTurbo Assemblerはそのファイルを間接ファイルとして扱い，コマンドライン上のどのオプションよりも優先的に処理されます。

あるプロジェクトに関するすべてのファイルが1つのディレクトリ内にあって，アセンブルは常に疑似浮動小数点命令（/**E**オプション）で行なう場合はこの機能は便利です。ファイルTASM.CFGの中にこのオプションを入れておけば，Turbo Assemblerを開始するたびに指定する必要がなくなります。

コンフィギュレーションファイルの内容は間接ファイルの内容とまったく同一です。ファイルには任意のコマンドラインオプションをいくつでも入れることができます。これらのオプションは1行のデータとして処理されます。

コンフィギュレーションファイルの内容は，コマンドラインのどの引数よりも先に処理されます。したがってコンフィギュレーションファイル内のオプションと逆の機能を持つオプションをコマンドラインから指定した場合は，コマンドラインのオプションが優先します。たとえばコンフィギュレーションファイルに

```
/A /E
```

という行があるときに

```
TASM /S /R MYFILE
```

というコマンドラインでTurbo Assemblerを開始したとします（MYFILEはソースファイル名です）。この場合は，ソースファイルはシーケンシャルセグメントオーダ（/**S**）および実浮動小数点命令（/**R**）でアセンブルされます。コンフィギュレーションファイル内の/**A**と/**E**オプションで指定したアルファベットセグメントオーダと疑似浮動小数点命令はともに無効となります。

第4章

アセンブラプログラミングの基本

これまではアセンブラプログラムの特長を説明してきました。この章では，アセンブラプログラムの基本的な事項について説明します。

最初に，アセンブラプログラムのもっとも基本的な事項に関して説明します。次にプログラム行のさまざまな構成要素と，それらをどのように組み合わせて使うのかについて説明します。順番に読み進めば，機械命令，疑似命令，アセンブラプログラムのメモリアクセス法がわかります。またセグメントを定義してそれを使用する方法やメモリ変数の定義と初期化について説明します。最後に一般的に使用される命令について説明します。

プログラムを作るには多くの基礎知識が必要ですが，この章を読めばプログラムを書くために最低限必要なことがわかります。この章では，章末に載せたワード数をカウントするプログラムを作るために必要な知識を提供します。

この章ではアセンブラプログラムのごく基本的な要素に触れるだけです。第5章「Turbo Assemblerプログラミング入門」と，第10章の「Turbo Assemblerの応用プログラミング」にアセンブリ言語のより詳しい解説を載せます。

アセンブラプログラムの構成要素と構造

8086のアセンブリ言語の基礎は説明しましたので，アセンブラプログラムを作る準備ができました。正常に動くアセンブラプログラムの必要最小限の事項について最初に説明します。簡単なアセンブラプログラムでもかなり多くの行数が必要です。次にアセンブラプログラムの例を示します。

```
    DOSSEG                                 ; セグメントオーダ疑似命令
    .MODEL    SMALL                        ; コードとデータをnearモデルにする
    .STACK    200h                         ; 512バイトのスタック
    .DATA                                  ; データセグメントの始まり
DisplayString  DB     13,10                ; 復帰改行(CR/LF)
ThreeChars     DB     3 DUP (?)            ; キーボードから読み込んだ3文字を
                                           ; 保存しておく変数
               DB     '$'                  ; ファンクション9でDisplayStringの文字を
                                           ; 表示するときの文字列の終わりを知らせる文字
    .CODE                                  ; コードセグメントの始まり
Begin:
    mov  ax,@data
    mov  ds,ax                             ; DSにデータセグメントをセット
    mov  bx,OFFSET ThreeChars              ; 最初の文字を保存しておくアドレスをbxにセット
    mov  ah,1                              ; キーボード入力ファンクション番号
    int  21h                               ; 次に押されたキーの文字コードを取り込む
    dec  al                                ; 文字コードから1引く
    mov  [bx],al                           ; 結果を保存
    inc  bx                                ; 次の文字に備えて保存場所のアドレスを
                                           ; インクリメントする
    int  21h                               ; 次に押されたキーの文字コードを取り込む
    dec  al                                ; 文字コードから1引く
    mov  [bx],al                           ; 結果を保存
    inc  bx                                ; 次の文字に備えて保存場所のアドレスを
                                           ; インクリメントする
    int  21h                               ; 次に押されたキーの文字コードを取り込む
    dec  al                                ; 文字コードから1引く
    mov  [bx],al                           ; 結果を保存
    mov  dx,OFFSET DisplayString           ; 保存してあった文字を文字列とし,
                                           ; その先頭アドレスをdxにセットする.
    mov  ah,9                              ; 文字列表示ファンクション番号
    int  21h                               ; 文字列を表示する
    mov  ah,4ch                            ; プロセス終了ファンクション番号
    int  21h                               ; プログラムの終了
    END  Begin                             ; ソースコードの終了を示し，プログラムの
                                           ; 開始場所を指示する疑似命令
```

このプログラムでは，簡略セグメント疑似命令**DOSSEG, .MODEL, .STACK, .DATA, .CODE,** および**END**疑似命令を使っています。セグメントの定義と使用法を指定するための簡略セグメント疑似命令は，どんなアセンブラプログラムでも必要です。また通常アセ

ンブラのコードは，**END**疑似命令で終了しなければなりません。この章ではセグメント疑似命令とENDなどの疑似命令を説明します。

疑似命令はアセンブラプログラムの大枠を指示する命令です。プログラムを実際に実行するためには，ソースコードの行が必要です。次に例を示します。

```
mov  [bx],al
```

あるいは

```
inc  dx
```

これらの行は，8086の命令セットに対応するニーモニックです（このマニュアルの最後の章で説明します)。機械命令や疑似命令を使う前には，アセンブラコードの行の書式を知ることが必要です。次に書式について説明します。

最初のサンプルプログラムがよくわからなければ，プログラムを入力して実行してみてください。プログラムを起動して**IBM**とタイプすると

```
HAL
```

と表示します。

このプログラムはキーボードから1文字読み込むたびにその値から1を引いて，その結果を表示するプログラムです。

行の書式

アセンブリ言語のソースコードの一般形式は次のとおりです。

<ラベル>　<機械命令/疑似命令>　<オペランド>　<; コメント>

<ラベル>は必要に応じて使用するシンボル名です。**<機械命令/疑似命令>**は機械命令または疑似命令のニーモニックです。**<オペランド>**は，それぞれの命令または疑似命令に対して必要な1つまたは複数の参照先メモリ，レジスタ，文字列です。オペランドは必要に応じて付けます。**<；コメント>**は必要に応じて書く注釈です。

次にアセンブリ言語コードのそれぞれの構成要素についてもう少し詳しく説明します。

ラベル

ラベルはプログラムの中で数値や文字列，メモリアドレスを参照するための名前です。ラベルによってメモリ変数，変数，特定の命令のアドレスに名前を付けることができます。5の階乗を計算する次のプログラムでは，多数のラベルを使っています。

```
        DOSSEG
        .MODEL   SMALL
        .STACK   200h
        .DATA
FactorialValue DW    ?
Factorial      DW    ?
        .CODE
FiveFactrial PROC
        mov  ax,@data
        mov  ds,ax
        mov  [FactorialValue],1
        mov  [factorial],2
        mov  cx,4
Factrialloop:
        mov  ax,[FactorialValue]
        mul  [factorial]
        mov  [FactorialValue],ax
        inc  [factorial]
        loop FactolialLoop
        ret
FiveFactrial  ENDP
        END
```

ラベル*FactorialValue*と*Factrial*はそれぞれ16ビット変数のアドレス名で，プログラムの

後半で参照されています。ラベル*FiveFactrial*はこのプログラムを他の部分から呼び出すためのサブルーチン（あるいは関数や手続き）名です。ラベル*FiveFactrialLoop*は,

```
mov  ax,[FactorialValue]
```

という命令があるメモリアドレスに付けられた名前です。

このプログラムの最後の方の**LOOP**命令でジャンプしてこの命令に戻ります。

ラベルには次のような文字が使えます。

```
A～Z    a～z    _    @    $    ?    0～9
```

MASMモード（第12章参照）では，ラベルの先頭文字のみにピリオド（.）も使えます。数字の0～9はラベルの先頭文字としては使えません。単独の や には特別な意味があるので，ユーザのシンボル名としては使わない方がよいでしょう。

それぞれのラベルは1回しか定義できません。つまり同一ラベルを異なる対象に使ってはなりません(ただし=疑似命令を使ったラベルの定義, Idealモードのサブルーチンやマクロの中のローカルラベルは例外です)。オペランド内ではラベルは何回でも使えます(参照できます)。

ラベルだけを1行に書くこともできます。つまり機械命令や疑似命令がない行も定義することができます。この場合，ラベルの値は次行の機械命令または疑似命令のアドレスになります。次に例を示します。

```
        .
        .
        .
        JMP   DoAddition
        .
        .
        .
DoAddition:
        add    ax,dx
        .
        .
        .
```

この例ではJMP命令の後でラベル*DoAddition*へ分岐して，**ADD AX, DX**を実行します。このプログラムは次のプログラムと同じです。

```
        .
        .
        .
        JMP   DoAddition
        .
        .
        .
DoAddition:  add    ax,dx
        .
        .
        .
```

ラベルだけを1行に書くことには2つの利点があります。1つにはアセンブラソースコードの書式とは無関係に長いラベルを付けることができます（2番目の例の**ADD AX, DX**が*DoAddition*の位置より右にずれて見にくくなっていることに注目してください)。また，ラベルと命令が2行にわけてあれば，そのラベルの右に容易に命令を追加することができます。さきほどの例に命令を追加してみます。

```
        .
        .
        .
        JMP   DoAddition
        .
        .
        .
DoAddition:  mov   dx,[MemVar]
        add    ax,dx
        .
        .
        .
```

2番目の例で同じことをやろうとしたら，まず*DoAddition*から**ADD AX, DX**を分離して，それから新たな文字を付け加えなければなりません。しかし*DoAddition*を単独に書いてあれば（1番目の例），*DoAddition*の行の右に文字を追加すればよいのです。

組み込みシンボルと同じラベルを使うことはできません。組み込みシンボルとは，たとえばレジスタ名（AX, BXなど）や演算子（**PTR, BYTE, WORD**など）として使われるものです。また疑似命令**IF*xxx***や**ERR*xxx***もラベルとして使うことはできません。Turbo Assemblerの予約シンボルは，特定の文脈で使える場合もあります。**NAME, INCLUDE, COMMENT**などがそれで，構造体のメンバ名としては使えますが汎用シンボルとしては使えません（構造体については第10章を参照してください)。

ラベル名として組み込みシンボル名を使わないのが無難です。次に，組み込みシンボルがラベルとして使えない例を示します。

```
bx    dw    0
PTR:
```

その理由はBXはレジスタ，**PTR**は演算子だからです。しかし次のラベルは大丈夫です。

```
Old_BX   DW    0
```

リファレンスガイドの第3章に疑似命令の一覧が，このマニュアルの第2章の表2.1にニーモニックの一覧があります。また8086のレジスタは第2章「Turbo Assemblerのプログラミング」に一覧があります。

使用できるラベルの例を次に示します。

```
MainLoop
calc_long_sum
Error0
iterate
Draw$Dot
Delay_100_milliseconds
```

疑似命令や命令ニーモニックのない単独のラベルや，命令ニーモニックのある行のラベルの後にコロン (:) を付けなければなりません。コロンは単に ラベルの終端を表わすものにすぎず，ラベルそのものではありません。次に例を示します。

```
        .
        .
        .
LoopTop:
        mov    al,[si]
        inc    si
        and    al,al
        jz     Done
        jmp    LoopTop
Done:   ret
        .
        .
        .
```

この例では，ラベル*LoopTop*と*Done*はコロンを付けて定義しています。参照するときはコロンを付けません。

その他のシンボルには通常コロンを付けません。この章のはじめのサンプルプログラムではコロンのないラベルを使っています。

ラベルは処理内容がわかるように付けます。次の2つの例を比べてみてください。

```
        .
        .
        .
        cmp   al,'a'
        jb    NotAlowerCaseLatter
        cmp   al,'z'
        ja    NotAlowerCaseLatter
        sub   al,20h                    ; 大文字への変換
NotAlowerCaseLatter:
        .
        .
        .
```

および

```
        .
        .
        .
        cmp   al,'a'
        jb    P1
        cmp   al,'z'
        ja    P1
        sub   al,20h                    ; 大文字への変換
P1:
        .
        .
        .
```

はじめのプログラム例のような説明的なラベルはたいへんわかりやすく，後の例のラベルはまるで暗号のようです。ラベルには下線記号も使えるので，*not_a_lower_case_letter*や*Not_A_Lower_Case_letter*とすることもできます。好みに応じて使ってください。

命令ニーモニックと疑似命令

アセンブラプログラムの行のキーポイントとになるフィールドは，**<機械命令/疑似命令>**のフィールドです。このフィールドには命令ニーモニックか疑似命令のどちらかが入ります。両者はまったく性質が異なります。

この章のはじめでも命令ニーモニックが登場しました。ニーモニックは8086が直接実行する機械語(マシンコード)に人間が憶えやすい名前を付けたものです。**MOV, ADD, MUL, JMP**はすべて命令ニーモニックで，それぞれ8086のデータ転送（data movement），加算（addition），乗算（multiplication），ジャンプ（branching）命令を表わします。

Turbo Assemblerは，それぞれの命令ニーモニックを直接マシンコードに翻訳します。つまりアセンブラプログラム内の命令ニーモニックは，対応するマシンコードに変換され

ます。

疑似命令は命令ニーモニックとは異なり，マシンコードを生成しません。アセンブルによって生成するコードの種類（8086, 80286, 80386など），セグメントの指定，リストファイルの作り方などをTurbo Assemblerに対して指示します。たまにその区別がはっきりしないことがありますが，命令ニーモニックは実際の8086のマシンコードを生成する命令で，疑似命令はアセンブラプログラミングを簡単にするためにTurbo Assemblerの高レベルの機能を指示する命令と考えることができます。

Turbo Assemblerで使える命令ニーモニックや疑似命令については，**リファレンスガイド**の第3章で説明します。プログラムにはいくつかの疑似命令が必要ですが，とりわけこの章の後半の「セグメント疑似命令」の節で説明するセグメント疑似命令はなくてはならないものです。**END**疑似命令も必要不可欠なもので，次に説明します。

END疑似命令

すべてのプログラムのソースコードの最後には**END**疑似命令が必要です。Turbo AssemblerはEND疑似命令以降の行はすべて無視します。もし**END**疑似命令がないとエラーが発生します。ファイルの終わりはプログラムの終わりであると考えがちですが，そうではありません。プログラムの終わりには必らず**END**を付けてください。

ENDはコードを生成しない典型的な疑似命令の1つです。次に例を示します。

```
        DOSSEG
        .MODEL   SMALL
        .STACK   200h
        .CODE
ProgramStart:
        mov  ah,4ch
        int  21h
END     ProgramStart
```

この例は何もしないでただちにDOSに戻るもっとも単純なアセンブラプログラムです。プログラムの最後に**END**疑似命令があることに注意してください。

この例の中で**END**と同じ行に*ProgramStart*があります。**END**はプログラムを終了するだけでなく，プログラムの実行開始位置を示す働きもあります。何らかの理由で，.EXEファイルの途中から実行したい場合があります。**END**はこのような場合を配慮しているわけです。たとえば次のコードをアセンブルしてリンクし，実行したとします。

```
        DOSSEG
        .MODEL   SMALL
        .STACK   200h
        .CODE
Delay:
        mov  cx,0
DelayLoop:
        loop  DelayLoop
        ret

ProgramStart:
        call  Deley                   ; ウェイトするため64Kのループを実行する
        mov   ah,4ch
        int   21h
END     ProgramStart
```

この例ではソースコードの先頭の*Delay*ラベルの**MOV CX, 0**からではなくて，**END**疑似命令で指示しているラベル*ProgramStart*の次の**call** *Delay*から実行を開始します。

1つのモジュールから成るプログラム（つまり1つのソースコードだけのファイル）では，一般に**END**疑似命令でプログラム開始アドレスを指示します。2つ以上のモジュールから成るプログラムでは，プログラムを開始するモジュール内の**END**疑似命令に対してのみ開始アドレスを指定し，他のモジュールの**END**には開始アドレスは指定しません。プログラムには開始アドレスが必要ですが，開始アドレスは1ヵ所で十分だからです。プログラムの開始アドレスが1つだけ指定されていることを確認してください（プログラムに2つの開始アドレスを指定すると，TLINKは最初に指定された開始アドレスを採用して他の開始アドレスは無視します）。

オペランド

命令ニーモニックと疑似命令によってTurbo Assemblerが行なうべきことを指示します。それに対してオペランドは，機械命令や疑似命令とレジスタ，パラメータ，メモリアドレスなどの関連を示します。たとえば**MOV**命令はそれ自身では意味がありません。転送元と転送先を指定しなければなりません。

機械命令にはオペランドが必要ないものもあれば，1つまたはそれ以上のオペランドが必要なものもあります。疑似命令では1行に入る限りいくつでもオペランドを指定できます。オペランドの正確な数は機械命令や疑似命令の種類によって異なります（命令によっては3つのオペランドを使うことがあります）。オペランドとして指定できる項目は，レジスタ，定数，ラベル，メモリ変数，文字列です。

1つだけオペランドを指定する場合は非常に明快で，その命令はそのオペランドのみを操

作します。次に例を示します。

```
push    ax
```

これはAXの値をスタックにプッシュします。オペランドを持たない命令はさらに明快です。では，ソースオペランド（転送元オペランド）とデスティネーションオペランド（転送先オペランド）の**2つ**のオペランドを持つ命令の場合はどうでしょうか 次に例を示します。

```
mov     ax,bx
```

この命令を実行すると，どちらのレジスタからどちらのレジスタへデータが動くのでしょう？

この命令の意味を〝AXの内容をBXに転送する〟と解釈する人もいるかもしれませんが，それは違います。この**MOV**命令はBXの内容をAXに転送します。MOV命令では，2つのオペランドの間のカンマ（,）を等号（=）に置き換えて，C（またはPascal）のように解釈すればよいでしょう。このやり方で**MOV**を書き換えてみると次のようになります。

```
ax = bx;
```

右側のオペランドがソースというのは確かにわかりづらいものですが，8086のアセンブリ言語ではこの原則が貫かれています。これはすぐに慣れます。

レジスタオペランド

レジスタは命令のオペランドとしてもっとも頻繁に使われます。レジスタはソースやデスティネーションとして利用されるだけでなく，場合によってはジャンプ先のアドレスとして使うこともあります。定数，ラベル，メモリ変数を使ってできることのほとんどは，レジスタを使って行なうことができます。一方レジスタオペランドしか使えない命令も数多くあります。

レジスタオペランドを使う命令の例を示します。

```
mov     di,ax
push    di
xchg    ah,dl
ror     dx,cl
in      al,dx
inc     si
```

レジスタオペランドは他の種類のオペランドと混ぜて使うことができます。

```
mov     al,1
add     [BaseCount],cx
cmp     si,[bx]
```

レジスタオペランドの使い方に関して説明すべきことはほとんどありません。レジスタオペランドを使うためには，命令のオペランドにレジスタ名を指定すればよいのです。2つのオペランドがある場合は，右側のオペランドがソースレジスタで，左側のオペランドがデスティネーションレジスタになります。また2つのソースが必要な命令なら両方ともソースレジスタになります。次に例を示します。

```
 .
 .
 .
mov     cx,1
mov     dx,2
sub     dx,cx
 .
 .
 .
```

この例はCXに1を，DXに2をそれぞれセットし，DXからCXを引いて結果の1をDXにストアします。**SUB**命令のCXは右側のオペランドなのでソースレジスタです。DXは左側のオペランドなので，もう1つのソースオペランドでもあり，デスティネーションオペランドでもあります。ところで**SUB**命令の働きは〝DXからCXを引く〟と表現できます。Cのコーディング方法で2つのオペランドの命令を表現してみると，**SUB**命令は次のように書くことができます。

```
dx -= cx;
```

Pascalなら，次のようになります。

```
dx := dx-cx;
```

定数オペランド

レジスタは変数の値を記憶しておくのに役立ちます。しかしオペランドに定数を必要とすることもあります。たとえばループでSIを4からカウントダウンして0にしたい場合は，次のようなプログラムが考えられます。

```
        .
        .
        .
CountFourLoop:
        .
        .
        .
        dec     si
        dec     si
        dec     si
        dec     si
        jnz     CountFourLoop
        .
        .
        .
```

このプログラムは，次のようにすればもっと簡単になります。

```
        .
        .
        .
CountFourLoop:
        .
        .
        .
        sub     si,4
        jnz     CountFourLoop
        .
        .
        .
```

文字も値を持っているので定数オペランドとして使えます。たとえば文字*A*は10進数で65の値を持っているので，次の2つの命令は同じ結果になります。

```
.
.
.
sub    al,'A'
sub    al,65
.
.
.
```

定数は10進数だけでなく2進数，8進数，16進数でも表記することができます。これらの使い方については，「ビット，バイト，基数」の節で後述します。

定数オペランドは2つのオペランドの左側に置くことはできません。定数はデスティネーションオペランドになることはできないからです。しかしソースオペランドとして値が必要な場合はどこにでも定数を記述できます。8086では定数の使用に関していくつかの制限があります。たとえば定数を直接スタックにプッシュすることはできません（これは8086と8088だけにある制限です)。たとえば5をプッシュするには，次のように2つの命令を使わ

なければなりません。

```
 .
 .
 .
mov     ax,5
push    ax
 .
 .
 .
```

定数が使用できない場合は，そのつど工夫する必要があります。幸いそのような命令は少ないうえに，万一定数を間違えて使用した場合はTurbo Assemblerが正しい方法を教えてくれます。

式

定数値が使用できるところではいつでも定数式が使えます。Turbo Assemblerでは，ネストしたカッコ，算術演算式，論理演算子，関係演算子，ラベルのセグメントやオフセットやメモリ変数のサイズ指定演算子などの任意の組み合わせの数式をサポートしています。

次に例を示します。

```
          .
          .
          .
MemVar  DB    0
NextVar DB    ?
          .
          .
          .
          mov     ax.SEG MemVar
          mov     ds,ax
          mov     bx,OFFSET MemVar+((3*2)-5)
          mov     BYTE PTR [bx],1
          .
          .
          .
```

この例では，**SEG**演算子を使って*MemVar*セグメントの値をAXにロードし，AXの値をDSに転送しています。次の，*****,**+**,**−**と**OFFSET**演算子を使った複雑な式は**OFFSET** *MemVar,1*と評価され，これは*NextVar*というアドレスを示します。最後は，**BYTE PTR**演算子を使ってBXで示される場所つまりNextVarに1を転送するバイト処理を実行しています。

式に関する重要な点は，すべての式は最終的に定数値にならなければならないことです。**OFFSET** *MemVar*は一定の値，つまりセグメント内の*MemVar*という名のオフセットで

す。*MemVar*にストアされている値が変っても*MemVar*というアドレス自身は変わりません。

通常，定数値は明確に決まっているため，Turbo Assemblerはアセンブル時に定数を評価します。Turbo Assemblerから見れば，**オフセット**の*MemVar*+2は5+2と同じことです。すべての式の構成要素は変化せずアセンブル時に決められ，式は最終的にはある定数値に置き換えられます。

式に使用できる演算子は次のとおりです。

```
<>, (), [], LENGTH, MASK, SIZE, WIDTH
. (構造体のメンバアクセス演算子)
HIGH, LOW
+, - (単項)
: (セグメントオーバーライド)
OFFSET, PTR, SEG, THIS, TYPE
*, /, MOD, SHL, SHR
+, - (二項)
EQ, GE, GT, LE, LT, NE
NOT
AND
OR, XOR
LARGE, SHORT, SMALL, .TYPE
```

多くの演算子は通常の算術式で使っているように使います。演算子についてはこの章でそのつど説明します。特定の演算子について疑問が生じたら，**リファレンスガイド**の第2章を参照してください。

ラベルオペランド

ラベルは多くの命令でオペランドとして使われます。適当な演算子があればラベルを定数値に変換できます。次に例を示します。

```
         .
         .
         .
MemWord DW  1
         .
         .
         .
         mov   al,SIZE MemWord
         .
         .
         .
```

この例ではメモリ変数*MemWord*のバイトサイズ，つまり2をALに転送します。この場合は最終セクションで示すようにラベルは式の一部として使われています。

ラベルは**CALL**や**JMP**命令の目的アドレス（分岐先アドレス）としても使います。次に例を示します。

```
        .
        .
        .
        cmp     ax,100
        ja      IsAbove100
        .
        .
        .
IsAbove100:
        .
        .
        .
```

この例では，AXが100を越えている場合は，**JA**命令によってオペランド*IsAbove100*で示されるアドレスにジャンプします。またラベルとして定数を指定して，分岐先のアドレスを示すこともできます。

ラベルはデータ操作命令のソースオペランドまたはデスティネーションオペランドとしてレジスタと同じように使うことができます。

```
        .
        .
        .
TempVar DW      ?
        .
        .
        .
        mov     [TempVar],ax
        sub     ax,[TempVar]
        .
        .
        .
```

この例では最初の命令でAXの内容をメモリ変数*TempVar*に転送し，次の命令でAXから*TempVar*の内容を引いています。

オペランドとしてラベルを使うことは，メモリアドレシングモードの重要な要素です。次にこれを説明します。

メモリアドレシングモード

メモリオペランドを使うときは，処理の対象となるデータのメモリ上の配置をあらかじめ知ることができません。したがって前の節で説明したように，適当なメモリ変数名を付けて指定します。たとえばメモリ変数*Assets*からメモリ変数*Debts*を引くには，次のようにします。

```
         .
         .
         .
Assets   DW     ?
Debts    DW     ?
         .
         .
         .
         mov    ax,[Debts]
         sub    [Assets],ax
         .
         .
         .
```

メモリアドレシングは見た目以上に便利なものです。ABCDEFGHIJKLMの文字を持つ文字列に*CharString*という名前を付けます。これは図4.1に示すようにデータセグメントのオフセット100から始まります。アドレス108にある9番目の文字を読むにはどうすればよいでしょうか？　C言語では次のように記述します。

（リスト1行 R04_38.lst）

Pascalならば次のようになります。

```
C := CharString[9];
```

ではアセンブラではどうでしょうか？　*CharString*にある文字は*A*なので，アセンブラではこのような方法で直接参照することはできません。

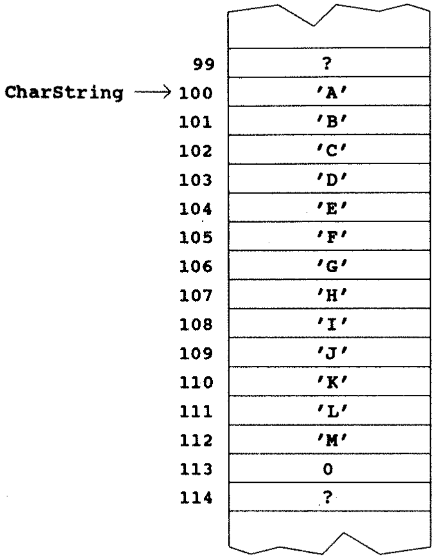


図4.1　文字列CharStringのメモリ配置

アセンブリ言語で文字列，配列，データバッファのアドレシングを取り扱うにはいくつかの方法があります。*CharString*の9番目の文字を読むもっとも簡単な方法は，次のようになります。

```
        .
        .
        .DATA
CharString    DB   'ABCDEFGHIJKLM',0
        .
        .
        .CODE
        .
        .


        .
        .
        mov   ax,@data
        mov   ds,ax
        mov   al,[CharString+8]
        .
        .
```

この場合*CharString*はオフセット100から始まるので，次のように記述することもできます。

```
        mov   al,[100+8]
```

Turbo Assemblerは大カッコで囲ったものはアドレスとして扱うため，上の例で*CharString*のオフセットに8を加えたものもアドレスとして評価します。これは次のように記述するのと同じです。

```
mov    al,[108]
```

このようすを図4.2に示します。

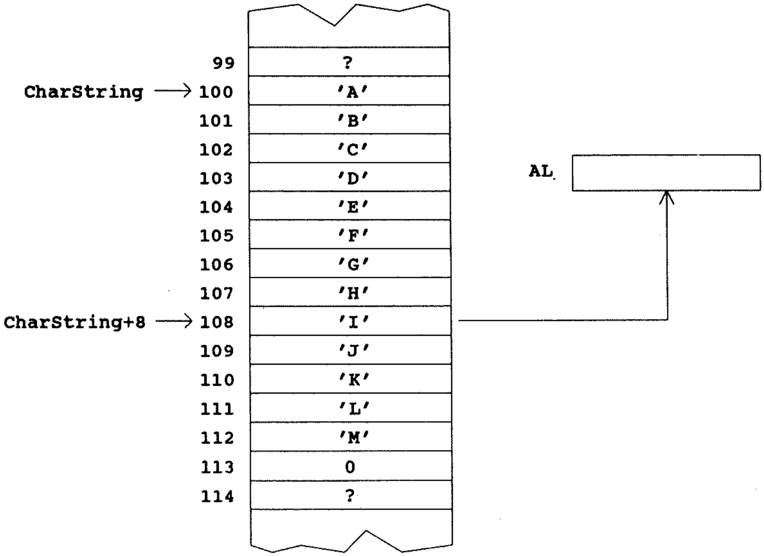


図4.2　文字列CharStringのアドレシング

次の命令はIdealモードでのみ正常に働きます。

```
mov    al,[100+8]
mov    al,[108]
```

MASMモードでは次のようになります。

```
mov    al,ds:[100+8]
mov    al,ds:[108]
```

名前だけまたは名前に定数を加えた形式で表記するこの種のアドレシングは，ダイレクトアドレシングと呼ばれます。**ダイレクトアドレシング**を使うと，常にメモリの同じアドレスにアクセスするので汎用性がありません。次に，より汎用性のあるメモリアクセスの方法を見ましょう。

*CharString*の9番目の文字をALレジスタに読み込む例を考えます。

```
.
.
.
mov   bx,OFFSET CharString+8
mov   al,[bx]
.
.
.
```

この例では9番目の文字を指し示すためにBXを使っています。最初に，*CharString*＋8のオフセット（**OFFSET**演算子はラベルが示すメモリのオフセット値をロードします）をBXにロードします（*CharString*＋8は**OFFSET**計算式で，アセンブル時に加算が行なわれます）。2番目の例では，BXで指し示したオフセットの値をALにロードします。

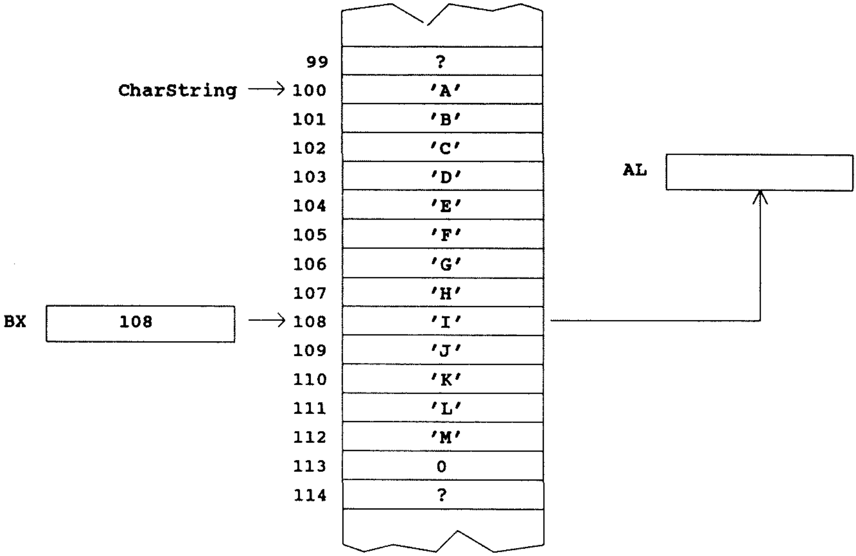


図4.3　BXを使ったCharStringのアドレス指定

大カッコ（[]）を使えば，BXそのものがソースオペランドにならずBXが指し示すメモリアドレスがソースオペランドになります。BXをメモリのポインタとして使うときはこの大カッコを忘れてはいけません。次に例を示します。

```
mov   ax,[bx]    ; BXが指し示すメモリの内容をAXに転送する
```

および

```
mov     ax,bx               ; BXの内容をAXに転送する
```

この2つの例はまったく違います。

ダイレクトオペランドでは同じ事をするのに1つの命令で済むのに,いったんメモリ変数のオフセットをBXにセットしてからBXをポインタにしてメモリにアクセスするのは厄介だと思うかもしれません。しかしダイレクトオペランドを使う命令とは違って,このようにメモリポインタとしてレジスタを使えば,プログラムの実行経過によって異なったメモリアドレスを指すことができるという利点があります。

たとえばヌルで終わる*CharString*の最後の文字を見つけたいときは,*CharString*の最初の文字から検索をはじめてヌルバイトを探し,1つ前に戻って最後の文字を読みます。このような処理では文字列の長さがはじめからわかっていないため,ダイレクトアドレシングを使うことはできません。BXをポインタレジスタにするテクニックを使います。

```
        .
        .
        .
        mov    bx,OFFSET CharString   ; 文字列の最初にポインタを設定
FindLastCharLoop:
        mov    al,[bx]                ; 文字列の次の文字をロード
        cmp    al,0                   ; 文字はゼロか?
        je     FoundEndOfString       ; ゼロなら1つ前の文字に
        inc    bx                     ; 次の文字にポインタを進める
        jmp    FindLastCharLoop       ; 次の文字をチェック
FoundEndOfString:
        dec    bx                     ; 最後の文字に戻す
        mov    al,[bx]                ; 文字列の最後の文字を取り出す
        .
        .
        .
```

メモリ上での文字やワードの検索,配列の処理,データのブロックコピーなどを行ないたい場合は,ポインタレジスタを固定値にできないことがわかります。

メモリポインタとして使うレジスタはBXの他にBP, SI, DIがあります。これらのレジスタは必要に応じて定数やラベルと組み合わせて使います。メモリオペランドの一般形式は次のとおりです。

```
[ベースレジスタ + インデックスレジスタ + 変位]
```

ここでベースレジスタはBXまたはBP,**インデックスレジスタ**はSIまたはDI,**変位**は16ビットの数値になるもので式やラベルも使えます。メモリオペランドを使った命令を実行するときは常にこの3つを組み合わせて使います。3つのうちどれか1つは必らず使わなければなりません(そうしないとアドレスを指定したことになりません)。メモリオペランドの形式を別な形式で示します。

BX		**SI**		
または	**＋**	**または**	**＋**	**変位**
BP		**DI**		
(ベース)		**(インデックス)**		

メモリアドレスの方法は，次のように16とおりになります。

- [**変位**]
- [bx]
- [si]
- [di]
- [bx+si]
- [bx+di]
- [bp+si]
- [bp+di]
- [bp+**変位**]
- [bx+**変位**]
- [si+**変位**]
- [di+**変位**]
- [bx+si+**変位**]
- [bx+di+**変位**]
- [bp+si+**変位**]
- [bp+di+**変位**]

変位は16ビットの定数として評価できるものです。

16種類のアドレッシングモードは一見複雑そうに見えますが，上のリストをよく見ると決まった要素の組み合わせにすぎないことがわかります。例として*CharString*の9番目の文字をALに転送する処理をいくつかのアドレッシングモードを使って示します。

```
        .
        .
        .DATA
CharString    DB    'ABCDEFGHIJKLM',0
        .
        .
        .CODE
        mov     ax,@data
        mov     ds,ax
        .
        .
        mov     si,OFFSET CharString+8
        mov     al,[si]
        .
        .
        mov     bx,OFFSET CharString
        mov     al,[bx+8]
        .
        .
        mov     si,8
        mov     al,[CharString+si]
        .
        .
```

```
        mov     bx,OFFSET CharString
        mov     di,8
        mov     al,[bx+di]
        .
        .
        mov     si,OFFSET CharString
        mov     bx,8
        mov     al,[si+bx]
        .
        .
        mov     bx,OFFSET CharString+8
        mov     si,7
        mov     al,[bx+si+1]
        .
        .
        mov     bx,3
        mov     si,5
        mov     al,[bx+CharString+si]
        .
        .
```

この例では，どの命令もすべて［*CharString*］＋8のアドレスを参照しています。

この例にはいくつかの興味ある点があります。1つには大カッコの中で使っている＋には特別の意味があることです。アセンブル時には，Turbo Assemblerは大カッコの中のすべての値を加算します。したがって

```
mov     [10+bx+1+si+100],cl
```

は次のようになります。

```
mov     [bx+si+111],cl
```

この命令を実行するとき（プログラム実行時）に，メモリアドレシングオペランドは加算されます。BXが25, SIが52のとき，**MOV**命令が実行されるとCLにはメモリアドレス25＋52＋111＝1888がセットされます。8086では，命令の実行時にベースレジスタ，インデックスレジスタ，変位を加算することに注意してください。つまり定数はアセンブル時に加算されますが，ベース，インデックス，変位は命令の実行時に加算されます。

これまでの例ではBPレジスタを使用しませんでした。BPはBXと少し異なる動作をします。BXはデータセグメントのオフセットとして使いますが，BPはスタックセグメントのオフセットとして使います。したがってデータセグメントにある*CharString*のアドレス指定には通常BPは使いません。

スタックセグメントのアドレシングのためにBPを使うことは第2章で説明しました。BPを使うときはスタックセグメントにデータが存在しなければならないことを除けば，BPは今までの例のBXと同じように使えるということを覚えておいてください。

（セグメントオーバーライドプレフィクスを使用すればBPはデータセグメントのアドレ

シングに**使えます**。BX, SI, DIは**セグメントオーバライドプレフィクス**を付ければ，スタックセグメント（SS），コードセグメント（CS），エクストラセグメント（ES）のアドレシングに使えます。セグメントオーバライドプレフィクスについては10章で説明します。しかし通常はセグメントオーバライドプレフィクスは必要ないため当面は無視します）。ダイレクトアドレスの大カッコは省略可能です。次の2つの例はまったく同じ処理を行ないます。

```
mov  al,[MemVar]
```

および

```
mov  al,MemVar
```

しかし混乱を避けるため，またできるだけわかりやすいプログラムを書くために，すべてのメモリ参照に大カッコを付けることを強くお勧めます。同様に大カッコがないほうが読みやすいと思う人もいるので，大カッコのないコードもあります。これは好みの問題です。どちらかのスタイルを選択して一貫してそれを使えば，プログラミング作業が円滑に行なえます。

次のようなメモリアドレシングの記述もあります。

```
mov  al,MemVar
```

または，

```
mov  al,CharString[bx][si]+1
```

これらの書式はどれも，いくつものメモリアドレシングの要素を＋のマークでつないで，大カッコでくくるのと同じです。つまり次の記述と同じです。

```
mov  al,[CharString+bx+si+1]
```

繰り返しますが，1つのスタイルを自分のものにしてそれを終始使うとよいでしょう。メモリへのレジスタポインタの大カッコは省略できません。大カッコを付けないと，たとえばBXはオペランドとして扱われ，オペランドへのポインタとみなされません。

コメント

最後にコメントフィールドについて説明します。コメントはプログラムの動作には関係ありません。つまり実行可能ファイルには何ら影響を与えないのですが，プログラミング

においては重要な要素となります。

プログラマの多くはC, Pascal, Prologなどの高級言語でのプログラミングを知っているといっても過言ではありません。それに比べてアセンブリ言語からプログラミングを学ぶ人はほとんどいません。アセンブリ言語を学ぶ人は，コメントの効用を折にふれアドバイスされたことでしょう。プログラムが複雑だったり，作ってから時間がたっていると，作った本人もプログラムがわからなくなってしまいます。したがってこのアドバイスは大変よいことです。

アセンブリ言語と比べて，Pascalのプログラムは実質的に可読性が高いプログラムです。Pascalのプログラムは理解しやすい制御構造をもち，強力な型指定変数，算術式，形式的なまたは任意の引数を伴う手続きや関数呼び出しを利用することができます。

これに対して，アセンブリ言語には組み込みの制御構造はありません。また強力ではあるがデータ型指定が不定である，変数を含む数式が使えない，独自のパラメータ受け渡し機構がないなどの特徴があります。端的に言えばアセンブラのプログラムは構造化されたメンテナンスの楽な高級言語のプログラムからは程遠いのです。しかしアセンブラプログラムが構造化できないとか，メンテナンスできないということではありません。アセンブラプログラムが本来もっているわかりにくさを少しでも軽減するために，コメント（およびサブルーチンやマクロ）を多用することが必要です。

アセンブラプログラムにコメントを付けるにはいろいろな方法があります。便利な方法の1つは，各命令の右マージンにコメントを書くことです。次に例を示します。

```
mov   [bx],al              ; 変更した文字をストア
```

こう書いた方が，次のように何もないよりわかりやすいでしょう。

```
mov   [bx],al
```

すべての行にコメントを付ける必要はありません。次のようなコメントは時間が経つと役に立たなくなります。

```
.
.
.
mov   al,1                  ; DOSのキーボードからの入力ファンクション
int   21h                   ; 次に押されたキーを読み取るファンクションコール
.
.
.
```

このようなコメントを付けるべきではないということではありません。コメントは次のように短かく要領よく書くのがよいでしょう。

```
.
.
.
mov   al,1
int   21h                 ; 次に押されたキーの読み取り
.
.
.
```

コメントの目的はプログラムの各部分を説明することではなく，プログラムの全体の流れを一目で把握できるようにすることです。

もう1つのコメントのテクニックは，プログラムのブロック単位でコメント行を挿入することです。この種のコメントには，行に付けるコメントよりも高レベルの処理概要を記述しておきます。次に例を示します。

```
        .
        .
        .
;
; 転送バッファのチェックサムバイトを計算
;
        mov   bx,OFFSET TransferBuffer
        mov   cx,TRANSFER_BUFFER_LENGTH
        sub   al,al                        ; チェックサムをクリア
Checksum:
        add   al,[bx]                      ; 現在のバイトの値を加える
        inc   bx                           ; 次のバイトをポイント
        loop  Checksum
        .
        .
        .
```

コメントはすべての行に必要なわけではありません。このブロックのコメントからは，BXに転送バッファのアドレスをセットし，CXにバッファの長さをセットすることがわかります。ここでのキーポイントは，コメントでこの7行のプログラムの働きを概説することです。したがって各行のコメントの重要度はより小さくなります。プログラムにざっと目を通す場合は，各行のコメントよりはブロック単位のコメントの方が役に立ちます。

高レベルのコメントの他のテクニックは，説明的なコメントを各サブルーチンの先頭に書くことです。コメントには，サブルーチンの機能，入力と出力の条件，レジスタの保存状態，サブルーチンを利用するときの諸注意などを記述します。次に例を示します。

```
;
; データバッファのチェックサムをバイトサイズで返すファンクション
;
; 入力:
;       DS:BX - バッファの先頭アドレス
;       CX    - バッファの長さ
;
; 出力:
;       AL    - バッファのチェックサム
;
; 使用レジスタ:
;       BX, CX
;
; 注意: バッファは64Kbを越えないこと
;        セグメントの境界を越えないこと
;
Checksum        PROC    NEAR
        sub     al,al                   ; チェックサムをゼロクリア
        add     al,[bx]                 ; 現在のバイトの値を加える
        inc     bx                      ; 次のバイトにポインタを進める
        loop    Checksum
        ret
Checksum        ENDP
```

こうしておけば，サブルーチンの動作をいったん確認したら以後このサブルーチンを検査する必要はほとんどありません。重要なことは，このサブルーチンをコールしたときに何が起こるかです。つまりこのサブルーチンをコールしたらどんな影響があるかがわかればよいのです。サブルーチンの先頭にこれらの詳細情報を記述しておけば，非常に役に立ちます。

この他にもコメントのテクニックはたくさんあります。各自のプログラミングスタイルにぴったりあうものを見つけてください。大切なのは，最初からコメントを付けることを習慣にして，コメントがプログラミングスタイルの中で重要な要素になるようにすることです。

セグメント疑似命令

この章と前の章で，セグメントとは何か，またそれがどのようにプログラムに影響を与えるかについて多くのページを使って説明しました。次に，Turbo Assemblerがどのようにセグメント内にデータやコードを配置するかについて説明します。

8086のアセンブリ言語では，セグメント指定が厄介な問題の1つです。そのため，Turbo Assemblerでは1種類だけではなく，2種類のセグメント制御疑似命令を備えています。1つは**簡略セグメント疑似命令**です。これはセグメント制御を比較的簡単に行なうことができ，高級言語とアセンブラモジュールをリンクする際に適した方法です。しかしTurbo Assemblerがもっているセグメント関連機能のいくつかしかサポートしていません。もう1つは**標準セグメント疑似命令**です。使用方法は複雑ですが，アセンブラを使う上で必要となるあらゆるセグメント制御に対応することができます。

以下に簡略セグメント疑似命令と標準セグメント疑似命令の両方を説明します。セグメント疑似命令の使い方の概要を示しますので，プログラムを書くために必要なことがわかります。セグメント疑似命令の詳細な説明は第10章「Turbo Assemblerの応用プログラミング」で説明します。

簡略セグメント疑似命令

簡略セグメント疑似命令は，**.STACK, .CODE, .DATA, .MODEL**です。このセクションではこれらの疑似命令を2つのグループにわけて説明します。はじめに**.STACK, .CODE, .DATA**について説明します。

.STACK, .CODE, .DATA

スタックセグメント，コードセグメント，データセグメントは，それぞれ，**.STACK, .CODE, .DATA**で定義します。**.STACK**はスタックのサイズを制御します。

次の例は200h（512）バイトのスタック領域を確保します。

```
.STACK    200h
```

スタックに関して行なうべきことはこれだけです。**.STACK**疑似命令をプログラムに入れたことを確かめておけば，あとはTurbo Assemblerがスタックを制御してくれます。通常

のプログラムでは200hが適切な大きさですが，スタックを頻繁に利用するプログラム，たとえば再帰プログラムではより大きなスタックを必要とします(このような例外的な**.STACK**の使い方については，第5章の後半にある「スタックの確保を忘れる，またはスタックが小さすぎる」節を参照してください)。

.CODEはプログラムのコードセグメントの始まりを指定します。すべての命令がコードセグメント内にあることをTurbo Assemblerに対して示すものです。Turbo Assemblerでは（標準セグメント疑似命令で）複数のコードセグメントを定義できます。その場合は命令をどのコードセグメントに配置するかを**.CODE**を使ってTurbo Assemblerに指示します。コードセグメントの定義はスタックセグメントの定義より単純で，オペランドを付けないで**.CODE**と記述しておけばよいのです。次に例を示します。

```
.
.
.
.CODE
sub     ax,ax                    ; アキュムレータをゼロにセット
mov     cx,100                   ; ループ回数をセット
.
.
.
```

.DATAはやや複雑です。**.DATA**はデータセグメントの先頭を指定します。このセグメントにはメモリ変数を置くことができます。次に例を示します。

```
        .
        .
        .
        .DATA
TopBoundary   DW   100
Counter       DW   ?
Errormessage  DB   0dh,0ah,'***Error***',0dh,0ah,'$'
        .
        .
        .
```

この例は単純です。**.DATA**で厄介なのは(実際には複雑ではないのですが)，**.DATA**で定義したセグメントにあるメモリにアクセスする前に，**DS**セグメントレジスタに**@data**シンボルをロードしておかなければならないことです。セグメントレジスタには汎用レジスタまたはメモリのどちらからでもデータをロードできますが，定数は直接ロードできません。したがって通常は次のように2行の命令を使って，DSセグメントレジスタに定数をロードします。

```
      .
      .
      .
      mov     ax,@data
      mov     ds,ax
      .
      .
      .
```

(AXレジスタのかわりに他の任意の汎用レジスタが使えます)。この例は**.DATA**疑似命令からはじまるデータセグメントをDSレジスタにセットしています。

次の例は，*DataString*に格納されているテキストを画面に表示するプログラムです。

```
        DOSSEG
        .MODEL  SMALL
        .STACK  200h
        .DATA
DataString      DB   'This text is in the data segment$'
        .CODE
ProgramStart:
        mov     bx,@data
        mov     ds,bx                  ; DSに.DATAセグメントをセット
        mov     dx,OFFSET DataString   ; .DATAセグメントでのDataStringの
                                       ; オフセットをDXにセット
        mov     ah,9                   ; 表示ファンクション番号
        int     21h                    ; 表示のファンクションコール
        mov     ah,4ch                 ; 終了ファンクション番号
        int     21h                    ; 終了のファンクションコール
        END     ProgramStart
```

.DATAで定義したセグメントをDSレジスタにセットする2つの命令がなければ，テキストの表示関数は正確に働きません。*DataString*は**.DATA**セグメントにあるため，DSにそのセグメントをセットしないとアクセスできません。DOSの関数で文字列を表示するには，文字列のセグメント オフセットのペアをDS：DXにセットします。**.DATA**セグメントをDSに，*DataString*のオフセットをDXにロードすれば，*DataString*へのセグメント オフセットのポインタを指定したことになります。ここで，DSは指定したのにCS, SS, ESは指定しなくてもよいのでしょうか？またESとは何でしょうか？

プログラムの実行時にDOSがCSを決定するので，CSはセットする必要はありません。本来，プログラム実行時にCSがセットされていないと，8086は最初の命令がどこにあるかわからずプログラムを実行しません。しかし実際は，CSに値がセットされていなければCSはプログラムが始まるときに自動的にセットされるため，必らずしもセットする必要はありません。

同様にSSはプログラムが始まる前にDOSがセットし，プログラム実行中は通常は変化しません。SSを変更することはもちろん可能です。しかしそのようなことが必要になるのはまれで，むやみに試みないほうが安全です。SSもCSと同様にプログラム開始時に自動的に

セットされるため，プログラムでセットする必要はありません。

しかしDSはまったくようすが違います。CSが命令を指し示し，SSはスタックを指し示し，DSはデータを指し示します。プログラムでは直接命令やスタックを変更しませんが，データは絶えずアクセスします。また同時に複数のセグメントのデータを取り扱うことがあります。8086では1Mバイトの範囲でメモリにアクセスできますが，一たびにアクセスできるのは（セグメントレジスタをベースとする）64Kバイトのブロック内だけです。

DSにあるセグメントをロードし，そのセグメントのデータをアクセスした後で別のブロックのデータをアクセスしたければ，DSに別のセグメントをロードします。このマニュアルに載っているようなスモールモデルやミディアムモデルのプログラムでは，データセグメントは1つしか必要ありません。しかしもっと大きなプログラムでは複数のデータセグメントを使うことがあります。またBIOSが使っているメモリなどのシステムメモリ領域にアクセスする場合は，DSにそのセグメント値をロードする必要があります。

したがってTurbo Assemblerでは，いつでもDSに任意のセグメントをセットすることができます。このような柔軟性があるかわりに，セグメント内のメモリにアクセスするためには，あらかじめそのセグメントをDSにセットしておかなければなりません。セットするには，通常 **@data** を使います。 **@data** は **.DATA** で始まるセグメントを示します。

ESセグメントレジスタはDSと同じようにセットできます。通常は，ESはまったく必要ありません。しかしESが指すセグメント内のメモリにアクセスしたいときは，最初にESに目的のセグメントをロードします。次に例を示します。このプログラムは，**.DATA** セグメントをESにロードし，ESを介してそのセグメント内の文字を表示します。

```
      DOSSEG
      .MODEL   SMALL
      .STACK   200h
      .DATA
OutputChar     DB     'B'
      .CODE
ProgramStart:
      mov  ax,@data
      mov  es,dx                          ; ESにデータセグメントをセット
      mov  bx,OFFSET OutputChar           ; OutputCharのオフセットをBXにセット
      mov  al,es:[bx]                     ; ESで示されるセグメントから文字を取り出す
      mov  ah,2                           ; DOSの表示ファンクション番号
      int  21h                            ; 表示のための割り込み
      mov  ah,4ch                         ; DOSの終了ファンクション番号
      int  21h                            ; 終了のための割り込み
      END  ProgramStart
```

DSの場合と同様，次のように2行の命令でESにセグメントをセットしていることに注意してください。

```
    .
    .
    .
    mov   ax,@data
    mov   es,dx                          ; ESにデータセグメントをセット
    .
    .
    .
```

この例では，DSよりもESを使った方がよい理由はありません。それどころかESを使用する場合は，ES:オーバライドプレフィクス（第10章参照）を使わなければならないのです。しかしDSに別のセグメントをセットして特にストリング命令を使っている場合は，ESにセグメントの1つをセットすると便利なことがあります。

DOSSEG

DOSSEG疑似命令は，アセンブラプログラム内のセグメントをマイクロソフト社のセグメントオーダ規定にしたがってグループ化します。このことについては今まで触れませんでした。ここでは，スタンドアロンのアセンブラプログラムの先頭には，**DOSSEG**を付けておくことだけを憶えておいてください。

高級言語とアセンブラモジュールをリンクする際は，高級言語は自動的にマイクロソフト社のセグメントオーダを採用するので，**DOSSEG**の指定はいりません。しかし**DOSSEG**を書いておいても別に問題はありません。むしろ現在有効なセグメントオーダを憶えておくのに役に立ちます。

したがってもっとも単純なアプローチは，(特別な理由のない限り)常に**DOSSEG**をプログラムの最初の行に入れておくことです。このようにすれば，一貫したセグメントオーダを指定することができます(**DOSSEG**の詳細は，**リファレンスガイド**の第3章を参照してください)。

.MODEL

.MODEL疑似命令は，簡略セグメント疑似命令を使うアセンブラモジュールのメモリモデルを指定します。farコードがCSとIPの両方をセットしてジャンプするのに対して，nearコードはIPレジスタのみをセットしてジャンプすることに注意してください。同様にnearデータはオフセットだけでアクセスされ，farデータはセグメント:オフセットのアドレスでアクセスされます。端的にいえば，far型では32ビットのセグメント:オフセットアドレスを使い，near型では16ビットのオフセットのみを使うということです。

使用できるメモリモデルは次のとおりです。

タイニィ	プログラムコードとデータの合計が64Kバイト以内で同一セグメント内に配置するモデル。コードもデータもnear型です。
スモール	プログラムコードは64Kバイト以内で1つのセグメント内に，データも64Kバイト以内で1つのセグメント内に配置するモデル。コードもデータもnear型です。
ミディアム	プログラムコードが64Kバイトを越え，データは64Kバイト以内で1つのセグメント内に配置するモデル。コードはfar型，データはnear型です。
コンパクト	プログラムコードは64Kバイト以内で1つのセグメント内に配置し，データは64Kバイトを越えるモデル。コードはnear型で，データはfar型です。
ラージ	プログラムコードもデータもともに64Kバイトを越えるモデル。ただし，1つのデータ配列が64Kバイトを越えることはありません。コードもデータもfar型です。
ヒュージ	プログラムコードもデータもともに64Kバイトを越え，しかもデータ配列の合計が64Kバイトを越えるモデル。コードもデータもfar型です。配列を指すポインタもfar型です。

アセンブラから見ると，ラージもヒュージもまったく同じです。ヒュージモデルは，64Kバイトを越えるデータ配列を無条件にサポートするわけではありません。

64Kを越えるコードまたはデータを必要とするアセンブラプログラムは少なく，ほとんどの場合はスモールモデルで十分です。できる限りスモールモデルを使ってください。farコード（コンパクト，ラージ，ヒュージの各モデル）はプログラムの実行速度が遅くなり，farデータ（コンパクト，ラージ，ヒュージの各モデル）を処理するにはアセンブラでは負担が大きくなります。

ここで説明したメモリモデルはTurbo C（および他の多くのコンパイラ）のメモリモデルに対応します。高級言語とアセンブラモジュールをリンクするときは，必らず**.MODEL**疑似命令を使ってください。**.MODEL**疑似命令を使えば，アセンブラのセグメント名が高級言語で使われているセグメント名と対応づけられ，手続き，関数，サブルーチン名として使われる**PROC**型のラベルを高級言語で使っている型（nearかfar）に合わせることができます。

簡略セグメント疑似命令を使うときは**.MODEL**が必要です。**.MODEL**がないと，Turbo Assemblerは**.CODE**と**.DATA**で定義されたセグメントをセットアップできません。**.MODEL**は**.CODE, .DATA, .STACK**より先に記述しなければなりません。

次に簡略セグメント疑似命令を使ったプログラムの概略を示します。

```
          DOSSEG
          .MODEL    SMALL
          .STACK    200h
          .DATA
MemVar    DW        0
          .
          .
          .
          .CODE
ProgramStart:
          mov   ax,@data
          mov   ds,ax
          mov   ax,[MemVar]
          .
          .
          .
          mov   ah,4ch
          int   21h
END       ProgramStart
```

その他の簡略セグメント疑似命令

一般にはあまり使われませんが，この他にもいくつかの簡略セグメント疑似命令があります。これはサイズの大きなまたは特に高度なアセンブラプログラムに限って必要になります。ここではその概要のみを説明します。詳しくは第10章を参照してください。

.DATA?は，初期化しないデータを含むデータセグメントを定義することを除いて**.DATA**と同じです。通常はアセンブラモジュールを高級言語とリンクするときに使います。

.FARDATAは，farデータセグメント，つまりすべてのモジュールで共有される標準の**@data**セグメントでないセグメントを定義するときに使います。**.FARDATA**は最大64Kバイトのfarデータセグメントを定義できます。**.DATA**疑似命令で指定したデータセグメントの名前が**@data**になるのと同様に，**.FARDATA**疑似命令で指定したデータセグメントの名前は**@fardata**になります。

.FARDATA?は，初期化していない**far**セグメントを定義することを除いて，**.FARDATA**と同様です。**.FARDATA**疑似命令と**@fardata**の組み合わせと同様，**.FARDATA?**疑似命令で指定したデータセグメントの名前は**@fardata?**になります。

.CONSTは，定数データを含むデータセグメント部分を定義します。この疑似命令は，高級言語とアセンブラコードをリンクするときに限って使います。

簡略セグメント疑似命令を使うときは，定義済みのラベルが使えます。**@FileName**はアセンブル中のファイル名です。**@curseg**はTurbo Assemblerが現在アセンブルしているセグメント名です。**@CodeSize**は，nearコードセグメントを持つメモリモデル(タイニィ，ス

モール，コンパクト）では0, farコードセグメントを持つメモリモデル（ミディアム，ラージ，ヒュージ）では1です。同様に，**@DataSize**は，nearデータセグメントを持つメモリモデル（タイニィ，スモール，ミディアム）では0，コンパクトまたはラージメモリモデルでは1，ヒュージモデルでは2です。

標準セグメント疑似命令

先ほどのサンプルプログラムを，標準セグメント疑似命令**SEGMENT, ENDS, ASSUME**を使って書き直すと次のようになります。

```
DGROUP  GROUP     _DATA,STACK
        ASSUME    CS:_TEXT, DS:_DATA, SS:STACK
STACK   SEGMENT   PARA STACK 'STACK'
        DB        200h  DUP (?)
STACK   ENDS
_DATA   SEGMENT   WORD PUBLIC 'DATA'
MemVar  DW        0
        .
        .
        .
_DATA   ENDS
_TEXT   SEGMENT   WORD PUBLIC 'CODE'
        .CODE
ProgramStart:
        mov  ax,@data
        mov  ds,ax
        mov  ax,[MemVar]
        .
        .
        .
        mov  ah,4ch
        int  21h
END     ProgramStart
```

簡略セグメント疑似命令がなぜ〝簡略〟と呼ばれるかおわかりでしょう。簡略セグメント疑似命令の機能の多くは，アセンブラモジュールを高級言語に簡単にリンクするためので，スタンドアロンのアセンブラプログラムには不必要です。次に〝*Hello, world*〟プログラムを標準セグメント疑似命令を使って書き直したプログラムを示します。

```
Stack    SEGMENT  PARA STACK 'STACK'
         DB       200h  DUP (?)
Stack    ENDS

Data     SEGMENT  WORD 'DATA'
HelloMessage    DB    'Hello,world',13,10,'$'
Data     ENDS

Code     SEGMENT  WORD 'CODE'
         ASSUME   CS:Code, DS:Data
ProgramStart:
         mov  ax,@data
         mov  ds,ax                    ; DSにデータセグメントをセット
         mov  dx,OFFSET HelloMessage ; HelloメッセージのポインタをDXにセット
         mov  ah,9                     ; 表示のDOSファンクション番号
         int  21h                      ; Helloメッセージの表示
         mov  ah,4ch                   ; プログラム終了のDOSファンクション番号
         int  21h                      ; プログラムの終了
Code     ENDS
END      ProgramStart
```

このサンプルプログラムは特に複雑ではありません。しかし標準セグメント疑似命令は簡略セグメント疑似命令よりも明らかに複雑です。

第10章で標準セグメント疑似命令を詳しく説明しています。この節では，それぞれの標準セグメント疑似命令の機能概要を説明します。

SEGMENT疑似命令はセグメントの始まりを示します。**SEGMENT**疑似命令の左のラベルはそのセグメント名です。次の例は*Cseg*という名前のセグメントの始まりです。

```
Cseg   SEGMENT
```

SEGMENT疑似命令は必要に応じて，さまざまなセグメント属性(バイト，ワード，ダブルワード，パラグラフ（16バイト)，ページ（256バイト）などのメモリ境界（アラインメントのタイプ）を指定できます。他の属性として，同名セグメントや同名クラスのセグメントどうしを結合する方法などがあります。

ENDS疑似命令はセグメントの終わりを定義します。次に例を示します。

```
Cseg   ENDS
```

この例では，ここより前で**SEGMENT**疑似命令で指定された*Cseg*という名前のセグメントの終わりを指定します。標準セグメント疑似命令を使う場合は，各セグメントの終わりを明示しなければなりません。

ASSUME疑似命令は，セグメントレジスタに現在どのセグメントがセットされているのかをTurbo Assemblerに指示します。**ASSUME CS:**という疑似命令は，標準セグメント疑似命令を使うプログラムでは必須です。なぜならば実行可能プログラムを作るためには，

Turbo Assemblerに対してコードセグメントを指示しなければならないからです。**ASSUME DS:**と**ASSUME ES:**によって，アクセスできるメモリアドレスをTurbo Assemblerに指示するために使います。

Turbo Assemblerは**ASSUME**を見つけると，現在のセグメントレジスタの設定でメモリ変数へのアクセスが有効かどうかをチェックします。次に例を示します。

```
Data1   SEGMENT  WORD 'DATA'
Var1    DW      0
Data1   ENDS

        .
        .
        .

Data2   SEGMENT  WORD 'DATA'
Var2    DW      0
Data2   ENDS

Code    SEGMENT  WORD 'CODE'
        ASSUME   CS:Code
ProgramStart:
        mov   ax,Data1
        mov   ds,ax              ; DSにDATA1をセット
        ASSUME DS:Data1
        mov   ax,[Var2]          ; AXにVar2をロード... Var2はData1セグメント内に
                                 ; はないので結果はエラー
        .
        .
        .
        mov   ah,4ch             ; DOSの終了ファンクションナンバ
        int   21h                ; プログラムの終了
Code    ENDS
END     ProgramStart
```

Turbo Assemblerは，このプログラムに対してエラーを出力します。なぜなら，DSに*Data1*セグメントがセットしてあるにもかかわらず，メモリ変数*Var2*をアクセスしようとしたからです。DSに*Data2*セグメントをセットしていないと，*Var2*をアクセスすることはできません。

Turbo Assemblerは，DSに*Data1*がセットされているかどうかをチェックできないことに注意してください。したがって**ASSUME**疑似命令を使ってTurbo AssemblerにDSの設定を知らせることが**必要**です。**ASSUME**は，ある時点のセグメントレジスタの設定をTurbo Assemblerに対して知らせるための手段です。Turbo Assemblerは，その設定にしたがってプログラムの正当性を判断します。

Turbo Assemblerはこのような間違いをすべて見つけることはできません。しかし名前付きメモリ変数はセグメントとの関係がはっきりしているので，*Var1*や*Var2*などの名前付きメモリ変数でメモリを参照するときは参照の正当性をチェックすることができます。次

の命令はどのセグメントをアクセスしようとしているのかわかりません。

```
mov   al,[bx]
```

このような場合, Turbo Assemblerはアクセスしようとしているセグメントが DSセグメントであると仮定します。

セグメントレジスタが名前付きセグメントを指していない場合は, **ASSUME**に**NOTHING**を指定して, Turbo Assemblerにその旨を知らせることができます。次に例を示します。

```
        .
        .
        .
mov   ax,0b800h
mov   ds,ax
ASSUME  DS:NOTHING
        .
        .
        .
```

この例ではDSにカラー テキスト画面アドレスをセットし, DSが名前付きセグメントを指し示していないことをTurbo Assemblerに知らせます (IBM PCの場合) カラー テキスト画面アドレスを指し示す方法は他にもあります。次に例を示します。

```
         .
         .
         .
ColorTextSeg     SEGMENT AT 0b800h
ColorTextMemory LABEL   BTYTE
ColorTextSeg     ENDS
         .
         .
         .
        mov   ax,ColorTextSeg
        mov   ds,ax
        ASSUME  DS:ColorTextSeg
         .
         .
         .
```

この例では, **SEGMENT**の後の**AT**疑似命令でセグメントの開始アドレスを指定しています (IBM PCの場合)。

ASSUME:に関する最後のポイントは, Turbo Assemblerは別のセグメントレジスタを使ってメモリにアクセスすることがあり得ることです。次に例を示します。

```
        .
        .
        .
Data1   SEGMENT  WORD 'DATA'
Var1    DW      0
Data1   ENDS

Data2   SEGMENT  WORD 'DATA'
Var2    DW      0
Data2   ENDS

Code    SEGMENT  WORD 'CODE'
        ASSUME   CS:Code
ProgramStart:
        mov   ax,Data1
        mov   ds,ax                    ; DSにDATA1をセット
        ASSUME DS:Data1
        mov   ax,[Var2]
        mov   es,ax                    ; ESにData2をセット
        ASSUME ES:Data2
        mov   ax,[Var2]                ; Var2をAXにセット. DSセグメント内にVer2が
                                       ; ないためESにData2セグメントをセット
        .
        .
        .
        mov   ah,4ch                   ; プログラム終了のDOSファンクション番号
        int   21h                      ; プログラムの終了
Code    ENDS
        END      ProgramStart
```

この例は，不正なメモリ参照における**ASSUME**の役割を示す先ほどのサンプルプログラムを変更したものです。この例ではエラーは出力しません。ミスは自動的に訂正されます。この例では，

```
mov    ax,[Var2]
```

を変更して，DSセグメントレジスタではなくESセグメントレジスタに基づいて*Var2*をアクセスできるようにしています。

この例では2つの**ASSUME**疑似命令で，DSをセグメント*Data1*に，ESをセグメント*Data2*にセットすることを指示します。次に**MOV**命令で*Data2*セグメントにある*Var2*をアクセスしようとします。DSに基づいて*Var2*をアクセスすることはできませんが，ESに対しては*Var2*をアクセスすることができます。したがって**MOV**命令の前に**セグメントオーバライドプレフィクス**と呼ばれる特別なコードを挿入して，DSではなくESを指定しています。

ASSUME疑似命令を使ってESとDSの現在の設定を指示すれば，Turbo Assemblerは名前を付けたメモリ変数にアクセスできるかどうかチェックします。アクセスできない場合は自動的に適切なセグメントを選択します。

セグメントオーバライドプレフィクスと標準セグメント疑似命令は，第10章で説明します。

簡略セグメント疑似命令と標準セグメント疑似命令

ここまでは簡略セグメント疑似命令と標準セグメント疑似命令の両方を説明しました。しかしどちらのセグメント疑似命令セットを使うべきなのかという疑問が残ります。この点はアセンブラプログラムの種類によって異なります。

高級言語とアセンブラモジュールをリンクする場合は，通常簡略セグメント疑似命令を使います。簡略セグメント疑似命令を使えば，セグメントの名前付けやメモリモデルの選択を高級言語のインターフェースと対応させて行なうことができます。

スモールモデルまたはミディアムモデルのスタンドアロンのアセンブラプログラムでは手軽に使えて読みやすいので，一般に簡略セグメント疑似命令を使います。

複数のセグメントを使った，モデルの混在した（nearモデルとfarモデルのコードやデータが混在した）ラージモデルのスタンドアロンのアセンブラプログラムでは，標準セグメント疑似命令を使います。なぜなら，標準セグメント疑似命令だけがセグメントの型，アラインメント，名前付け，セグメントの結合方法の指定などができるからです。

標準セグメント疑似命令を使わないとセグメントの定義ができない場合を除いては，原則として簡略セグメント疑似命令を使ってください。

データの配置

セグメントの指定方法を説明しましたので，次にセグメントへデータをストアする方法を説明します。スタックセグメントは問題ありません。ここにはスタックが積まれます。**PUSH, POP**またはBPレジスタを使わないとスタックにはアクセスできません。コードセグメントは，プログラムの命令ニーモニックで生成した命令コードだけのセグメントです。残りはデータセグメントです。Turbo Assemblerには，データセグメント内で変数を定義するさまざまな方法があります。定義の際に，特定の値に初期化することも初期化しないでおくこともできます。Turbo Assemblerで定義できるデータの種類を理解するために，アセンブラのデータ型について説明します。

ビット，バイト，基数

コンピュータが記憶するデータの最小単位は**ビット**です。1ビットは1か0の値をとります。ビット単独ではあまり役に立ちません。8086ではビットを単独で取り扱うことはせず，8ビットからなる**バイト**単位でデータを取り扱います。

1ビットは2進数の1桁に相当します。1バイトは8ビットの2進数です。8ビット2進数で表現できる数は次のとおりです。

```
2の0乗：        1
2の1乗：        2
2の2乗：        4
2の3乗：        8
2の4乗：       16
2の5乗：       32
2の6乗：       64
2の7乗：    + 128
           -------
              255
```

1バイトでは0〜255の範囲の数を表わすことができます。

8086の8ビットレジスタ（AL, AH, BL, BH, CL, AH, DL, DH）はそれぞれ1バイトを保持します。8086の1,000,000を越える各メモリアドレスには，それぞれ1バイトをストアすることができます。

コンピュータに使われる文字セット（大文字も小文字も含めたアルファベット，0〜9の数字，グラフィック文字，特殊記号，カナ文字，句読点などを含む）には，全部で256種類

の文字があります。文字セットは，1文字が1バイトに対応するように設計されています。

1バイトは8086が扱える最小単位であり，1文字，0～255の符号なし数値，-128～+127の符号付き数値などを表わすことができます。1バイトだけでは，整数や浮動小数点数，あるいはメモリポインタを扱うには明らかに不十分です。

次に大きいデータ単位は**ワード**（16ビット）です。1ワードは2バイト（16ビット）です。ワードは2つのバイトを連続して配置したもので，8086のメモリでは500,000以上のワードのメモリ空間があります。8086の16ビットレジスタ（AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, ES, SS, IP，フラグレジスタ）はどれも1ワードを保持できます。1ワードには16ビットの2進数を保持できます。16ビットで表現できる数は次のとおりです。

```
2の 0乗:        1
2の 1乗:        2
2の 2乗:        4
2の 3乗:        8
2の 4乗:       16
2の 5乗:       32
2の 6乗:       64
2の 7乗:      128
2の 8乗:      256
2の 9乗:      512
2の10乗:     1024
2の11乗:     2048
2の12乗:     4096
2の13乗:     8192
2の14乗:    16384
2の15乗:  + 32768
         ---------
            65536
```

これが16ビットで表現できる符号なし整数の最大値です。符号付き整数(-32768～+32767の範囲）も同様にワードで表わせます。

ワードはサイズが16ビットなので，セグメント内のオフセットのアドレスを表わすことができ，メモリポインタとして使うことができます。ワードサイズのBX, BP, SI, DIレジスタがメモリポインタとして使えたことを思い出してください。

32ビット（4バイト）で表わせる値は**ダブルワード**（*doublewords*または*dwords*）です。8086では32ビットの整数を直接取り扱えないため，**ADC**や**SBB**命令で32ビットの整数計算をする場合は2回の16ビット操作によって計算します。ダブルワードは，符号なし整数0～4,294,967,295，および符号付き整数+-2,147,483,648～ 2,147,483,647を扱うことができます。

8086では**LDS**と**LES**命令でセグメント オフセットのダブルワードをセグメントレジスタと汎用レジスタの両方にロードできますが，ダブルワードを直接サポートしているとはいえません。単精度の浮動小数点数はダブルワードで格納します（単精度の浮動小数点数は4バイトを必要とし，10^{-38}～10^{38}の値を表わすことができます)。

倍精度の浮動小数点数は8バイトを必要とします。このような64ビットの値は**クオッドワード**(*quadwords*)と呼びます。8086はクオッドワードはサポートしていません。しかし8087数値演算コプロセッサはクオッドワードを基本的なデータ型として扱っています（単精度の浮動小数点数は10^{-308}～10^{308}の値を表わし，最大有効桁数として16桁の精度を持ちます)。

Turbo Assemblerは，一時的な（中間的な）浮動小数点数である10バイトのデータサイズをサポートします。この10バイトのデータは2進化10進数（BCD）としても使われます。この場合は1バイトで2つの10進数を表わします。

8086ではワードやダブルワードを**低位バイトから先に**メモリに保存します。たとえばアドレス0にはワードの7-0ビットの部分を，アドレス1には15-8ビットの部分を保存します。図4.4のようになります。

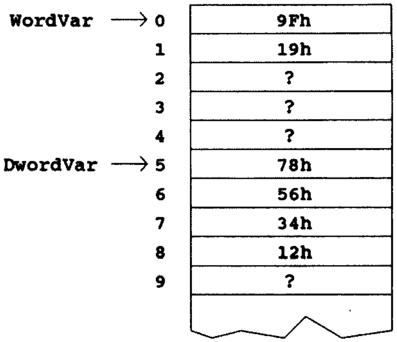


図4.4　199Fhの値をもつワード変数WordVarと
12345678hの値をもつダブルワード変数DwordVar

同様にダブルワードをアドレス5に保存すると，7-0ビットはアドレス5に，15-8ビットはアドレス6に，23-16ビットはアドレス7に，31-24ビットはアドレス8に保存されます。少し奇妙にみえるかもしれませんが，これがiAPx86ファミリのプロセッサに共通のストア方式です。

10進数，2進数，8進数，16進数

アセンブリ言語のデータ型を説明しましたので，次はその表記法を説明します。10進数（10を基数とする数）は日常生活で使っているように表記すればよいので簡単です。次に例を示します。

```
mov    cx,100                  ; ループのカウンタを100にセット
```

Turbo Assemblerは，特に指定のない限り数値は10進数とみなします。しかし10進法はアセンブリ言語に適した数値の表記法ではありません。コンピュータが2進法に基づいているためです。

したがってアセンブラプログラムでは2進数を使うことが合理的でしょう。Turbo Assemblerで2進数を表わす場合は，数値の最後にbを付けます（2進数は0と1で記述されます）。たとえば10進数の5を2進数で表記すれば101bになります。

2進法による表記には，数値が長くなりすぎて使いにくいという問題があります。次の表のように各桁が0か1しか使えないからです。

10進	2進	8進	16進
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17
24	11000	30	18
25	11001	31	19
26	11010	32	1A
:	:	:	:
256	10000000	400	100
:	:	:	:
4096	100000000000	10000	1000
:	:	:	:
65536	1000000000000000	200000	10000

次に2進数の例を示します。

```
mov    cx,1100100b                  ; ループのカウンタに10進数の100をセット
```

ワードやダブルワードの2進数ともなると，読むのも書くのも容易ではありません。

2進数，8進数，16進数はアセンブリ言語のプログラミングの基本です。このような表記に慣れていない方には適当な本で勉強することをぜひお勧めします。

2進数を基礎とするコンピュータに適するだけでなく，扱いやすいコンパクトな表記とし

て**8進数**と**16進数**があります。

8進数では0～7の数字を使い，1桁は3ビットを表わします。図4.5は2進数の001100100b（10進で100）を3ビットごとにわけて8進数の144oに変換するようすを示します。

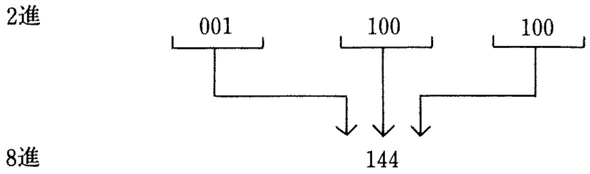


図4.5　2進数001100100（10進数の100）から8進数144への変換

8進数は2進数に比べて1/3の長さで済みます。8進数を使って先ほどの例を書き直すと次のようになります。

```
mov    cx,144o                    ; ループのカウンタに10進数の100をセット
```

接尾語（サフィクス）*o*が8進数を表わしていることに注意してください。サフィクスがゼロとまぎらわしいと感じたら，かわりに*q*を使うこともできます。

8進法は非常に便利なためコンピュータのある分野では広く使われています。しかし通常のプログラマは8進数よりも16進数をよく使います。

16進数では1桁で16の値をとることができます。16進数で0から数えると次のようになります。

```
0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  10  .....
```

9以降の6つの数字には*A*～*F*を当てます（小文字の*a*～*f*も同様に使えます）。数字にアルファベットを使うのは奇妙に思われますが，16個の数を表わすためには仕方ありません。図4.6は2 進数の01100100b（10進数で100）を4ビットごとにわけて16進数の64hに変換するようすを示します。

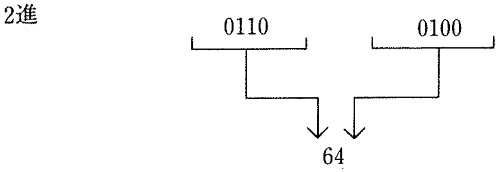


図4.6　2進数の01100100bから16進数の64hへの変換

16進数は図4.6に示すように4ビットで1桁を作る方法です。したがって16進数は2進数に比べて1/4の長さで表記できます。オフセットやその他のワード値は4桁の16進数で表わすことができます。16進数では，さきほどの例を16進数を使って書き直すと次のようになります。

```
mov     cx,64h                  ; ループのカウンタに10進数の100をセット
```

16進数はサフィクス*h*を付けて表わします。16進数は0〜9で始まらなければなりません。これはたとえば16進数のBAD4はラベルの***BAD4h***と間違えられることがあるからです。次に16進数のBAD4hと，ラベルの***BAD4h***を混在させた例を示します。

```
        .
        .
        .
        .DATA
BAD4h   DW    0                 ; ラベルのBAD4h
        .
        .
        .
        .CODE
        mov    ax,0BAD4h        ; AXに16進数をロード
                                ; (0で始まっていることが定数であることを示す)
        .
        .
        .
        mov    ax,BAD4h         ; AXにメモリ変数BAD4hの値をロード
                                ; (0で始まっていないことがラベルであることを示す)
        .
        .
        .
```

一般に，数字の0〜9で始まるオペランドは定数値です。

浮動小数点数はこれらのどの方法でも表せません。浮動小数点数は仮数と指数を使って表記します。次に例を示します。

```
  1.1
-12.45
  1.0E12
252.123E-6
```

Turbo Assemblerは仮数/指数形式を浮動小数点に従った2進数に変換します。必要なら16進数の後に*r*サフィクスを付けて，IEEE形式やマイクロソフトバイナリフォームで浮動小数点数を記述することもできます。

実数は，後述する**DD, DQ, DT**などの疑似命令でのみ使えます。*r*サフィクスを使う場合はそのデータ型を初期化するために必要な16進数の最大桁（必要なら先頭の0も）を明確に指定しなければなりません。次に例を示します。

```
DD  40000000r                    ; 2.0(8桁)
DQ  0C014cccccccccccccr          ; -5.2(16桁と先頭の0)
DT  4037D529AE9E86000000r        ; 1.2E17(20桁)
```

一般に仮数/指数形式を使うのが便利です。

数値が10進数であることを表わすためにサフィクス*d*を使うこともできます。10進数に*d*サフィクスを付ける場合は，10進数以外の形式をサフィクスなしの形式として指定した場合です。この指定は後述の**.RADIX**疑似命令で行ないます。

引用符('または")で囲んだデータは文字定数を表わします。文字の値はASCIIコードの値です。次の例は文字*A*をALレジスタにロードします。

```
mov   al,65
mov   al,41h
mov   al,'A'
mov   al,"A"
```

これまで説明した表記法のうち，2進数，8進数，16進数と文字はどこにでも定数として使えます。次に例を示します。

```
mov   ax,1001b
add   cx,5bh
sub   [Count],117o
and   al,
mov   al,'A'
```

浮動小数点数は**DD, DQ, DT**を使わないと表わせません。BCD（2進化10進数）は**DT**でのみ使えます。

デフォルト基数の選択

通常は10進数がもっとも慣れた表記法であるため，サフィクスなしの数値として10進数がデフォルトとなっています。しかしサフィクスなしの数値として他の表記法を割り当てたいこともあります。そのような場合は，**.RADIX**疑似命令が必要になります(*Radix*は数値記法における基数という意味です)。

.RADIXは，サフィクスなしの数字をどの表記とみなすかを指定します。次に例を示します。

```
.RADIX  16
```

この例は基数として16を選択します。つまり16進表記をデフォルトとします。次に**.RADIX**疑似命令の効果を示します。

```
    .
    .
    .
.RADIX  16          ; デフォルトを16進数に設定
.mov    ax,100      ; 100hつまり10進数の256
.RADIX  10          ; デフォルトを10進数に設定
sub     ax,100      ; 10進数で100を引く．結果は10進数の256-100=156
.RADIX  2           ; デフォルトを2進数に設定
add     ax,100      ; 2進数で100bつまり10進数で4を加える．結果は10進数の156+4=160
    .
    .
    .
```

.RADIXを使うと，基数として2, 8, 10, 16のどれかを選択することができます。なお，どの表記法を選択する場合でも**.RADIX**のオペランドは常に10進数で指定します。つまり．**RADIX**疑似命令による指定は，**.RADIX**疑似命令のオペランド自身の表記法には影響を与えません。

.RADIX疑似命令を使う場所はまったく自由です。どの表記法がデフォルトとして選択されていても，**DD, DQ, DT**に関しては，サフィクスなしの数値は常に10進数であると認識されます。次に例を示します。

```
    .
    .
    .
.RADIX  16
DD      1E7
    .
    .
    .
```

この例では1E7は1×10^7の意味で，1E7hではありません。ところで.RADIX疑似命令でたとえ16進数を指定してあっても，すべての16進数にサフィクス*h*を付けることを勧めます。その理由は，2進数や10進数に付けるサフィクス*b*や*d*は，16進数の数字でもあるからです。**.RADIX 16**が指定されている場合，129dや101bなどの数字はどのように評価されるでしょう？

このような場合，Turbo Assemblerはサフィクスに注目します。129dは10進数の129, 101bは2 進数の101つまり10進数の5とみなします。つまり，たとえ**.RADIX 16**が指定されていても，*b*や*d*で終わる16進数にはサフィクス*h*を付けなければならないということです。16進数には常にサフィクス*h*を付けるようにすれば確実です。したがって通常は**.RADIX 16**を使う必要はほとんどありません。

初期化済みデータの定義

Turbo Assemblerでメモリ変数を定義する方法について説明します。最初に初期化済みデータの定義について説明します。

データ定義の疑似命令は**DB, DW, DD, DF, DP, DQ, DT**で，それぞれ次のようなサイズのメモリ変数を定義します。

DB	1バイト
DW	2バイト=1ワード
DD	4バイト=1ダブルワード
DF, DP	6バイト=1farポインタワード（386）
DQ	8バイト=1クオッドワード
DT	10バイト

次に例を示します。

```
        .
        .
        .
        .DATA
ByteVar     DB    'Z'              ; 1バイト
WordVar     DW    101b             ; 2バイト(1ワード)
DwordVar    DD    2BFh             ; 4バイト(1ダブルワード)
QwordVar    DQ    307o             ; 8バイト(1クオッドワード)
TwordVar    DT    100              ; 10バイト
        .
        .
        .
        mov   ah,2                 ; DOSのディスプレイ表示ファンクション番号
        mov   dl,[ByteVar]         ; 表示する文字をロード
        int   21h                  ; DOS表示のファンクションコール
        .
        .
        .
        add   WORD PTR [DwordVar],ax
        adc   WORD PTR [DwordVar+2],dx
        .
        .
        .
```

この例では5つの初期化済みメモリ変数を定義し，これらの変数の使い方を示しています。

配列の初期化

1つのデータ定義疑似命令で複数の値を定義できます。次に例を示します。

```
SamleArray   DW   0, 1, 2, 3, 4
```

この例は図4.7に示すように,5個のワードサイズの要素を持つ*SampleArray*という配列を作ります。データ定義疑似命令では，1行に入る限りいくつでも値を定義できます。

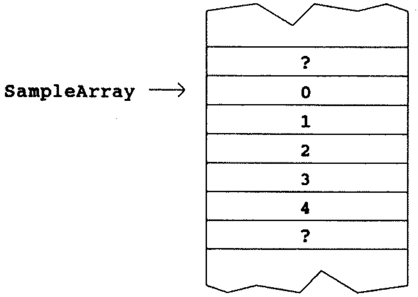


図4.7　5個の要素を持つ配列

たとえすべての要素が1行に入りきらなければ次の行に続けます。データ定義疑似命令には，ラベルを付ける必要はありません。次に例を示します。

```
        .
        .
        .
SquaresArray   DD   0, 1, 4, 9, 16
               DD   25, 36, 49, 64, 81
               DD   100, 121, 144, 169, 196
        .
        .
        .
```

この例では1～15の整数の平方数を要素とする*SquaresArray*という名前のダブルワードサイズの配列が作られます。

Turbo Assemblerでは,**DUP**演算子を使えばメモリのブロックを任意の値に初期化することができます。次に例を示します。

```
BlankArrey    DW   100h DUP (0)
```

この例では,ゼロで初期化した256ワードから成る*BlankArray*という名前の配列が作られます。次も同じような例です。

```
ArreyOfA       DW    92 DUP ('A')
```

この例では，文字Aで初期化した92バイトから成るArray0fAという名前の配列が作られます。

文字列の初期化

データ定義疑似命令は文字に対しても使えます。文字列を初期化する場合は次のようにします。

```
String   DB   'A', 'B', 'C', 'D'
```

この例は，Turbo Assemblerでは次のように簡略化することができます。

```
String   DB   'ABCD'
```

0で終わるCスタイルの文字列を使う場合は，その終わりに0を付けます。また復帰(CR)や改行（LF）コードが必要なら，それらを同様にして挿入できます。次の例は，テキストの最後に復帰（CR）と改行（LF）が付き，0で終わる文字列です。

```
HelloString   DB   'Hello, world',0dh,0ah,0
```

画面上で改行させるためには，復帰（CR）と改行（LF）のコードが必要です。次に例を示します。

```
        DOSSEG
        .MODEL    SMALL
        .STACK    200h
        .DATA
String1 DB       'Line1','$'
String2 DB       'Line2','$'
String3 DB       'Line3','$'
        .CODE
ProgramStart:
        mov   ax,@data
        mov   ds,ax
        mov   ah,9                        ; DOSの表示ファンクション番号
        mov   dx,OFFSET String1           ; 表示する文字列
        int   21h                         ; 表示のDOSファンクションの実行
        mov   ah,9                        ; 表示のDOSのファンクション番号
        mov   dx,OFFSET String2           ; 表示する文字列
        int   21h                         ; 表示のDOSファンクションの実行
        mov   ah,9                        ; DOSの表示ファンクション番号
        mov   dx,OFFSET String3           ; 表示する文字列
        int   21h                         ; 表示のDOSファンクションの実行
        mov   ah,4ch
        int   21h
        END   ProgramStart
```

この例を実行すると，次のように画面に表示されます。

```
Line1Line2Line3
```

次に復帰（CR)/改行（LF）コードを各文字列の終わりに付け加えます。

```
String1 DB      'Line1',0dh,0ah,'$'
String2 DB      'Line2',0dh,0ah,'$'
String3 DB      'Line3',0dh,0ah,'$'
```

この場合は，表示は次のようになります。

```
Line1
Line2
Line3
```

式とラベルによる初期化

初期化した変数の初期値は1つですが，初期値は1つの数値である必要はありません。次のような式も使えます。

```
TestVar   DW  ((942/2)+1)
```

さらにラベルを使うこともできます。

```
          .
          .
          .
          .DATA
Buffer            DW    16 DUP (0)
BufferPointer     DW    Buffer
          .
          .
          .
```

データ定義疑似命令のオペランドにラベルを使う場合は，設定されるのはラベル自身の値であって，そのアドレスにストアされている値ではありません。上の例では，*BufferPointer*の初期値は**.DATA**セグメント内のオフセット*Buffer*であり，*Buffer*に保存されている値(0)ではありません。これは，**OFFSET** *Buffer*を使って*BufferPointer*を初期化することと同じです。次の2つの命令はともにAXに同じ値，つまりオフセット*Buffer*をロードします。

```
mov   ax,OFFSET Buffer
```

および

```
mov   ax,[BufferPointer]
```

ラベルはデータ定義式にも使えます。次のプログラムは，変数*WordArrayLength*を*WordArray*のサイズ（バイト長）に初期化します。

```
        .
        .
        .
        .DATA
WordArray        DW      50 DUP (0)
WordArrayEnd     LABEL   WORD
WordArrayLength  DW      (WordArrayEnd - WordArray)
        .
        .
        .
```

*WordArray*の長さをバイトではなくワードで計算したい場合は，次にようにバイトの長さを2で割ります。

```
WordArrayLength DW      (WordArrayEnd - WordArray)/2
```

初期化しないデータの定義

メモリ変数に初期値を割り当てる必要がない場合があります。次の例は，キーボードからタイプした10文字を*KeyBuffer*という名前の配列に読み込みむプログラムです。

```
        .
        .
        .
        mov    cx,10                      ; 読み込む文字の数
        mov    bx,OFFSET KeyBuffer        ; 文字を保存するKeyBuufer
GetKeyLoop:
        mov    ah,1                       ; キー入力のDOSファンクション番号
        int    21h                        ; キーの読み込み
        mov    [bx],al                    ; 文字保存
        inc    bx                         ; 次のストア先にポインタ移動
        loop   GetKeyLoop
        .
        .
        .
```

*KeyBuffer*を次のように初期化しておくこともできます。

```
KeyBuufer    DB  10  DUP (0)
```

しかし，*KeyBuffer*に初期値を設定しても，*GetKeyLoop*でただちに上書きされるため意味

がありません。メモリ変数を初期化しないで定義する場合は疑問符（？）を使います。

疑問符はTurbo Assemblerに格納場所を確保することを指示しますが，初期化は行ないません。さきほどの例で初期値を指定しないで*KeyBuffer*を定義するには，次のようにします。

```
KeyBuufer    DB  10  DUP (?)
```

この例では*KeyBuffer*というラベルから，初期化されない*10*バイトが確保されます。

初期化していないメモリ変数を使う場合は，必らずプログラムで初期値を設定しなければならないことはいうまでもありません。先の例では*KeyBuffer*には初期値が定義されていないため，*KeyBuffer*の値をすぐに読み込んで使うのは誤りです。

名前が付けられたメモリアドレス

これまでは，**DB**などのデータ定義疑似命令の前にラベルを付けてメモリアドレスに名前を付ける方法を説明しました。メモリを確保しないでメモリアドレスに名前を付けるには，**LABEL**疑似命令を使います。

LABELは，ラベル名を指定し，データを定義しないでその型を指定します。次の例は先の例で示した配列*KeyBuffer*を定義する別の方法です。

```
         .
         .
         .
KeyBuffer    LABEL   BYTE
         DB    10   DUP   (?)
         .
         .
         .
```

LABELで定義できるラベルの型には次のようなものがあります。

BYTE	**PWORD**	**FAR**
WORD	**QWORD**	**PROC**
DWORD	**TBYTE**	**UNKNOWN**
FWORD	**NEAR**	

BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTEは，それぞれ1, 2, 4, 6, 8, 10バイトのデータです。次の例はメモリ変数を2バイトとして初期化し，ワードとしてアクセスするプログラムです。

```
        .
        .
        .
        .DATA
WordVar LABEL    WORD
        DB       1,2
        .
        .
        .
        .CODE
        .
        .
        .
        mov      ax,[WordVar]
        .
        .
        .
```

このプログラムを実行すると，ALに1(*WordVar*の最初のバイト)がロードされ，AHには2（*WordVar*の2番目のバイト）がロードされます。

NEARと**FAR**は，コールまたはジャンプの型を選択するために使います。次に例を示します。

```
        .
        .
        .
        .CODE
        .
        .
        .
FarLabel        LABEL   FAR
NearLabel       LABEL   NEAR
        mov     ax,1
        .
        .
        .
        jmp     FarLabel
        .
        .
        .
        jmp     NearLabel
        .
        .
        .
```

最初の**JMP**命令のラベルは**FAR**ラベルなので，このジャンプはfarジャンプです(CSとIPの両方を設定します)。次のJMP命令のラベルは**NEAR**ラベルなので，このジャンプはnearジャンプです(IPだけを設定します)。*near*ラベルも*far*ラベルも**MOV**命令と同じアドレスに記述してあるのに，異なった方法でその場所に分岐できることに注意してください。

簡略セグメント疑似命令を使っている場合は，**PROC**疑似命令を使って現在のコードモデルに対して適切なサイズ（nearまたはfar）のラベルを定義するのが便利です。メモリモデルがタイニィ，スモール，コンパクトのどれかの場合は，**LABEL PROC**は**LABEL NEAR**と同じです。メモリモデルがミディアム，ラージ，ヒュージのどれかの場合は，**LABEL PROC**は**LABEL FAR**と同じです。したがって，メモリモデルを変更すると自動的にラベルも変わります。次に例を示します。

```
        DOSSEG
        .MODEL   SMALL
        .
        .
        .
        .CODE
        .
        .
        .
EntryPoint    LABEL    FAR
        .
        .
        .
```

この例では*EntryPoint*はnearですが，メモリモデルをラージに変えると*EntryPoint*はfarになります。通常は，メモリモデルの変更に伴って変わるエントリポイントを定義するには，**LABEL**よりもむしろ**PROC**疑似命令（この章の終わり近くの「サブルーチン」の節で説明します）を使います。ただしサブルーチンに複数のエントリポイントを必要とする場合は，**PROC**の他に**LABEL**も必要となります。

最後に**LABEL UNKNOWN**について説明します。**UNKNOWN**は，使うラベルのデータ型が明らかでないときに便利です。**UNKNOWN**はCの**void**型のようなものです。**UNKNOWN**の例として，あるときはバイトで，またあるときはワードでアクセスされるメモリ変数*TempVar*を考えます。次に**LABEL UNKNOWN**を使った例を示します。

```
        .
        .
        .
        .DATA
TempVar LABEL  UNKNOWN
        DB     ?,?
        .
        .
        .
        .CODE
        .
        .
        .
        mov    [TempVar],ax
        .
        .
        .
        add    dl,[TempVar]
        .
        .
        .
        .
```

データの転送

ここまでは，アセンブリ言語の特徴，アセンブラの基本的な概念，アセンブラプログラムの構造について説明してきました。基本的な説明はこれで終了し，次にアセンブラの命令に焦点を当てます。機械命令はアセンブラプログラムの要素であり，実際に8086を働かせるものです。はじめにもっとも基本的なアセンブラの命令である，データ転送命令について説明します。

MOVは8086のデータ転送命令です。**MOV**はソースオペランドの内容をデスティネーションオペランドにコピーするため，**MOV**というニーモニックよりむしろ**COPY**の方が適切かもしれません。ソースオペランドは元のままです。次に例を示します。

```
.
.
.
mov     ax,0
mov     bx,9
mov     ax,bx
.
.
.
```

この例では下図のように，まず定数0をAXにストアし，次に定数9をBXにストアし，最後にBXの内容をAXにコピーします。

mov ax, 0**実行後**：	AX	0
	BX	?
mov bx, 9**実行後**：	AX	0
	BX	9
mov ax, bx**実行後**：	AX	9
	BX	9

定数9はBXからAXに移されているのではなく，コピーされていることに注意してください。

オペランドにセグメントレジスタが指定されている場合を除いて，**MOV**に任意の組合わせのオペランドを指定できます（これについては少し後の「セグメントレジスタのアクセス」の節で説明します)。定数，定数値になる式，汎用レジスタ，メモリアドレス（いずれも「メモリアドレシングモード」の節で説明したアドレシングモードでアクセスされます）が**MOV**のソースオペランド（右側オペランド）になります。汎用レジスタやメモリアドレスが**MOV**のデスティネーションオペランド（左側オペランド）になります。

データサイズの選択

アセンブリ言語では，**MOV**命令によってバイトやワードの値をコピーすることができます。Turbo Assemblerがアクセスするデータサイズを決める方法を次に説明します。

多くの場合，**MOV**命令のオペランドはデータのサイズを指定します。オペランドがレジスタならば，そのレジスタのサイズがデータサイズになります。次の例ではデータサイズは明らかです。

```
.
.
.
mov al,1                ; バイトサイズ
mov dx,si               ; ワードサイズ
mov bx,[di]             ; ワードサイズ
mov [bp+si+2],al        ; バイトサイズ
.
.
.
```

同様に名前の付いたメモリアドレスも固有のサイズを持っています。したがって次の命令のデータサイズも明らかです。

```
        .
        .
        .
        .DATA
TestChar        DB    ?
TempPointer     DW    TestChar
        .
        .
        .
        .CODE
        .
        .
        .
        mov     [TestChar],'A'
        mov     [TempPointer],0
        .
        .
        .
```

しかし，たまに**MOV**命令のデータサイズが決まらないこともあります。次の命令では，バイトサイズの値なのかワードサイズの値なのか不明です。

```
mov    [bx],1
```

Turbo Assemblerは，このような命令のアセンブルはできない旨のメッセージを出力します。これはワードサイズの値を一時的にバイトとしてアクセスしたい場合やその逆の場合に便利です。

Turbo Assemblerでは，**WORD PTR**と**BYTE PTR**の演算子によって，あらかじめ決められたサイズとは無関係にデータサイズをフレキシブルに決めることができます。**WORD PTR**はメモリオペランドをワードサイズとして扱います。**BYTE PTR**はメモリオペランドをバイトサイズとして扱います。次の例は，先ほどの例のワードサイズの定数1をBXが指すワードに転送します。

```
mov    WORD PTR  [bx],1
```

次の例はバイトサイズの定数1をBXが指し示すバイトに転送します。

```
mov    BYTE PTR  [bx],1
```

WORD PTRや**BYTE PTR**をレジスタに適用しても意味がありません。レジスタのサイズは常に一定だからです。たとえ指定しても，**WORD PTR**と**BYTE PTR**は無視されます。**WORD PTR**と**BYTE PTR**を定数に対して使った場合も無視されます。定数はデスティネーションオペランドと常に同じサイズを持つからです。

WORD PTRと**BYTE PTR**には別の使い方もあります。名前の付いたメモリ変数に対して，一時的に異なったデータサイズを指定できます。それが何の役に立つのでしょう？

では次の例を見てください。

```
        .
        .
        .
        .DATA
Source1 DD      12345h
Source2 dd      54321h
        .
        .
        .
        .CODE
        .
        .
        .
        mov     ax,WORD PTR [SOURCE1]       ; ソース1の下位ワードをロード
        mov     dx,WORD PTR [Source1+2]     ; ソース1の上位ワードをロード
        add     ax,WORD PTR [Source2]       ; ソース2の下位ワードに加える
        add     dx,WORD PTR [Source2+2]     ; ソース2の上位ワードに加える
        mov     WORD PTR [Sum],ax           ; 下位ワードの和をストア
        mov     WORD PTR [Sum],dx           ; 上位ワードの和をストア
```

この例で使っている変数はすべて倍長整数かダブルワードです。8086ではダブルワードどうしを直接加算することはできないため，ワードサイズの式に分解してから計算を行なわなくてはなりません。変数自体がダブルワードでも，**WORD PTR**を使えば*Source1, Source2, Sum*をそれぞれワードとしてアクセスすることができるのです。

FAR PTRと**NEAR PTR**の演算子は，データサイズには直接影響を与えません。これは**WORD PTR**や**BYTE PTR**に似ています。**FAR PTR**はジャンプやコールの分岐先ラベルをfarラベルとして扱います(ジャンプやコールを実行する際はCSとIPをともにセットします)。一方**NEAR PTR**では，ラベルをnearラベルとして扱います(ジャンプやコールを実行する際はIPのみをセットします)。

符号付きデータと符号なしデータ

符号付きの数字も符号なしの数字も一連の2進数でできています。符号付きデータと符号なしデータの区別はアセンブラプログラマが行ない，8086自身は行ないません。たとえば0FFFFhという値は，ユーザのプログラムによって65,535と-1のどちらにも解釈できます。0FFFFhが-1である理由は，次のようにこの数値に1を加えてみればわかります。

```
  .
  .
  .
  mov   ax,0ffffh
  mov   ax,1
  .
  .
  .
```

この結果は0になります。-1に1を加えた結果と同じです。

ADD命令は，オペランドが符号付きか符号なしかにかかわらず使えます。次に0FFFFhから1を引く場合を考えます。

```
  .
  .
  .
  mov   ax,0ffffh
  sub   ax,1
  .
  .
  .
```

結果は0FFFEhとなります。これは（符号なしの数字ならば）65,534または（符号付きの数字ならば）-2です。

もしこれがわかりにくかったら，このマニュアルの最後に載せてある本のリストの中から一冊を読んで，8086が符号付きの数字を扱うための***2の補数***について学ぶとよいでしょう。符号付きの数値の計算は，アセンブラプログラマがアセンブラを理解するために非常に役に立ちますが，残念なことにここでは十分解説するスペースがありません。ここでは符号付きの数字でも符号なしの数字でも**ADD, SUB, ADC, SBB**が同じように使え，符号付きの加算や減算のための特別な命令がないということを憶えておいてください。後で説明するように，符号の有無はデータサイズの変換や条件ジャンプで問題となります。

データサイズの変換

ワードをバイトに変換したり逆にバイトをワードに変換しなければならない場合があります。データサイズの変換では数値の符号の有無が問題となります。

はじめにワードをバイトに変換する方法を示します。これは簡単で，ワードの上位バイトを捨ててしまえばよいのです。次に例を示します。

```
        .
        .
        .
        mov     ax,5
        mov     bl,al
        .
        .
        .
```

この例ではAXのワード値5をバイト値5に変換してBLにストアします。もちろん変換する値はバイトに適したものでなくてはなりません。たとえば100hをバイトに変換しようとして次のようにしても無駄です。下位バイトの0がALにストアされるだけです。

```
        .
        .
        .
        mov     dx,100h
        mov     al,dl
        .
        .
        .
```

符号なしのバイト値をワードに変換するには,ワードの上位バイトを0にすればよいのです。次に例を示します。

```
        .
        .
        .
        mov     cl,12
        mov     al,cl
        mov     ah,0
        .
        .
        .
```

この例ではCLにストアされていた符号なしのバイト値12は,符号なしのワード値12に変換されます。

符号付きバイトをワードに変換するのはもう少し複雑です。8086にはそのための特別な命令**CBW**があります。**CBW**はALの符号付きバイトをAXの符号付きワードに変換します。次の例ではDHの符号付きバイト-1をDXの符号付きワード-1に変換します。

```
	.
	.
	.
	mov	dh,-1
	mov	al,dh
	cbw
	mov	dx,ax
	.
	.
	.
```

8086には**CWD**という，AXの符号付きワードをDX：AX(上位ワードがDX)のダブルワードに変換する特別な命令もあります。次の例ではAXの符号付きワード＋10,000をDX：AXの符号付きダブルワード＋10,000に変換します。

```
	.
	.
	.
	mov	ax,10000
	cwd
	.
	.
	.
```

符号なしワードの値は，ダブルワードの上位ワードの値をゼロにすることによって符号なしダブルワードの値に変換できます。

セグメントレジスタのアクセス

MOV命令はセグメントレジスタへ値を転送したり，セグメントレジスタから値を転送したりできます。しかしこれは特殊な使い方です。**MOV**命令の2つのオペランドの一方がセグメントレジスタであれば，他方のオペランドは汎用レジスタかメモリアドレスでなければなりません。定数を直接セグメントレジスタにロードしたり，セグメントレジスタの値を別のセグメントレジスタに直接コピーすることはできません。

セグメント名は定数ですから，セグメントレジスタにロードするには汎用レジスタかメモリ変数を使わなければなりません。たとえば，ESに**.DATA**セグメントをセットするには2つの方法があります。次に例を示します。

```
        .
        .
        .
        .DATA
DataSeg DW    @data
        .
        .
        .
        .CODE
        .
        .
        .
        mov     ax,@data
        mov     es,ax
        .
        .
        .
        mov     es,[DataSeg]
        .
        .
        .
```

次のようにロードすることはできません。

```
mov     es,@data                ; これはできない
```

セグメントレジスタの内容を別のセグメントレジスタにコピーするには，汎用レジスタかメモリを経由させなければなりません。次の2例はともにCSの内容をDSにコピーします。

```
.
.
.
mov     ax,cs
mov     ds,ax
.
.
.
```

および

```
.
.
.
push    cs
pop     ds
.
.
.
```

上の例の方が実行速度は速いのですが，コードのサイズは下の例の方が小さくなります。セグメントレジスタを使う場合は，**MOV**命令以外にも制限を受けるものがあります。ほと

んどの命令ではセグメントレジスタをオペランドとしては使えません。セグメントレジスタに対して行なえることは，スタックにプッシュしたりポップすることぐらいです。加算，減算，論理演算，比較演算には使えません。

スタックのアクセス

スタックとはLIFO(last-in, first-out)，つまり後入れ先出しでデータを出し入れできるスタックセグメント内の領域であることはすでに述べました。スタックの先頭アドレスは常にSPが指し示しています。BPをベースポインタとするメモリアドレシングモードを使えば，**MOV**命令でスタック上のデータをアクセスすることができます。次に例を示します。

```
mov    ax,[bp+4]
```

この例はBP＋4で表わされるスタックセグメント内のオフセットのワードの内容をAXにロードします（BP経由のスタックアクセスについては第2章を参照してください)。

スタックは，通常**PUSH**や**POP**命令でアクセスされます。**PUSH**はオペランドをスタックの先頭にストアします。**POP**はスタックの先頭の値を取り出し，オペランドにストアします。次に例を示します。

```
.
.
.
mov     ax,1
push    ax
pop     bx
.
.
.
```

この例では，AXの値（1）をスタックの先頭にプッシュして，次にスタックの先頭から1をポップしてBXにストアします。

データの交換

XCHG命令は2つのオペランドの内容を交換します。他の命令を使うと3つの命令が必要になるような操作にたいへん便利です。次の例ではAXとDXの内容が入れ替わります。

```
xchg    ax,dx
```

これは次の3つの命令を実行するのと同じです。

```
.
.
.
push    ax
mov     ax,dx
pop     dx
.
.
.
```

I/O（入出力）

ここまでは，8086のメモリアドレスと定数値またはレジスタとの間のデータ転送について説明してきました。8086には独立したアドレス空間があります。このアドレスは，入出力アドレスまたはI/Oアドレスと呼ばれます。65,536のI/OアドレスつまりI/Oポートは，ディスクドライブ，ディスプレイ，キーボード，プリンタなどのデバイスに対するコントロールチャネルとデータチャネルの役割を果たします。

MOVなどの8086の命令の多くは，メモリ空間のオペランドにのみアクセスできます。I/Oポートにアクセスできるのは**IN**と**OUT**という2つの命令だけです。

IN命令は指定したI/OポートのデータをALまたはAXにロードします。I/Oポートのアドレスをソースオペランドとして指定するには，次の2つの方法のどちらかを使います。I/Oポートのアドレスが256（100h）より小さい場合は，次のようにI/Oアドレスを指定します。

```
in    al,41h
```

この例ではI/Oポート41hのバイトデータを**AL**にロードします。

次のようにDXにI/Oポートアドレスを指定してデータを読み取ることもできます。

```
.
.
.
mov    dx,41h
in     al,dx
.
.
.
```

なぜDXをI/Oポインタとして使うのでしょうか？　理由の1つは，I/Oポートのアドレスが255より大きい場合はDXを**使わなければならないから**です。もう1つの理由は，DXを使

えばI/Oポートへのアドレシングに柔軟性をもたせることができます。たとえばDXにI/Oポートのポインタをセットしておいて，それをサブルーチンに渡すことができます。

IN命令の文法を間違えないようにしてください。ALとAX**のみ**がデスティネーションオペランドとして使えます。同様にDXと256未満の定数だけがソースオペランドとして有効です。次のような命令は使えません。

```
in    bh,si                  ; これは実行できない
```

OUT命令は**IN**命令と同様です。ただしAXまたはALがソースオペランドとなり，DXまたは256未満の定数によって指定したI/Oポートがデスティネーションオペランドとなります。次の例はI/Oポート3B4hに0Fhを出力します。

```
.
.
.
mov   dx,3b4h
mov   al,0fh
out   dx,al
.
.
.
```

演算

コンピュータはデータ転送処理に多くの時間を費やします。データ転送は非常に重要な要素ですが，算術演算や論理演算によるデータ処理も同様に重要な要素です。次に8086の算術演算と論理演算について説明します。

算術演算

コンピュータは数値演算にすべての時間を費やしているわけではありませんが，必要があればそうさせることは**可能**です。それには，スプレッドシートやデータベースのプログラム，エンジニアリングパッケージなどをPC上で実行すればよいのです。こうすれば8086はパワフルな計算機になります。本当にそうでしょうか？

その答は何とも言えません。8086上で走るソフトウェアには優れた演算能力があるのは事実ですが，8086自身はごく基本的な計算能力しか備えていません。第1に，8086はその卓越した機能は別として，浮動小数点演算（5.2や1.03E17といった実数計算をいいます）の命令はサポートしていません。浮動小数点演算は8087数値演算コプロセッサの仕事です（8086のプログラムでは浮動小数点演算はできないわけではありません）。もちろん8087がなくてもスプレッドシートを走らせることはできます。8086上で浮動小数点演算を行なうためには，一連のシフト命令，加算命令，テスト命令が必要となり，8087の高速命令に比較して実行速度が遅くなります。

また8086には16ビットを越えるオペランドを直接扱える算術演算命令や論理演算命令がありません。

8086が標準でサポートする演算の種類は，8ビットと16ビットの符号付きおよび符号なし加減乗除の命令，および高速で特殊なインクリメント/デクリメント命令です。16ビットを越える値の演算を行なうためにはいくつもの命令が必要となります。8086では16ビットを越える数値の加算と減算を行なうことができます。

加算と減算

ADD命令と**SUB**命令については，これまでのサンプルプログラムの中ですでに説明しました。この2つの命令でかなり多くの処理ができます。**ADD**はソースオペランド（右側）の内容とデスティネーションオペランドを加え，その結果をデスティネーションオペランド

にストアします。SUBはデスティネーションオペランドからソースオペランドを引くことを除いてADDと同じです。次に例を示します。

```
          .
          .
          .
          .DATA
BaseVal   DW    99
Adjust    DW    10
          .
          .
          .
          .CODE
          .
          .
          .
          mov     dx,[BaseVal]
          add     dx,11
          sub     dx,[Adjust]
          .
          .
          .
```

この例では*BaseVal*の定数99をDXに格納し，次にこれに定数11を加算します。その結果DXの値は110になります。さらに*Adjust*に格納してある定数10をDXから引き，結果の100をDXに格納します。

32ビットのオペランド

ADDと**SUB**には8ビットまたは16ビットのオペランドが使えます。もし32ビットオペランドで加算や減算を行ないたければ，ワードサイズの演算に分解したうえで，**ADC**や**SBB**を使わなければなりません。

2つのオペランドの加算を行なうと8086はステータスをキャリフラグ(フラグレジスタのCビット)にセットします。キャリフラグはデスティネーションオペランドが桁あふれしている（加算の結果がデスティネーションオペランドに納まっていない）ことを示します。10進法の加算の桁上がりには慣れていることでしょう。たとえば90に10を足すと3桁目に繰り上がります。

```
   90
+  10
-----
  100
```

16進数の加算では次のようになります。

```
  FFFF
+    1
 10000
```

計算結果の下位ワードはゼロになり，計算結果の10000hは16ビットに納まらないため，キャリは1になります。

ADCは，キャリフラグ（前回の加算によってセットされているはず）を考慮して計算するという点を除いて**ADD**と同じです。ワードよりも大きな2つの値の加算を行なうときは，先に数値の下位ワードを**ADD**で加算して，次に**ADC**命令を1回または複数回使って残りのワードを加算し，最上位ワードは最後に加算します。たとえば，CX：BXに格納されているダブルワードの値とDX：AXに格納されているダブルワードの値を加える場合は，次のようになります。

```
.
.
.
add     ax,bx
sdc     dx,cx
.
.
.
```

次の例は，*DoubleLong1*のクオッドワードを*DoubleLonge2*のクオッドワードに加算します。

```
.
.
.
mov     ax,[DoubleLong1]
add     [DoubleLong2],ax
mov     ax,[DoubleLong1+2]
adc     [DoubleLong2+2],ax
mov     ax,[DoubleLong1+6]
adc     [DoubleLong2+6],ax
.
.
.
```

SBBは，**ADC**と同様な方法で減算を行ないます。**SBB**はそれまでの減算で生じた借りをすべて考慮にいれます。たとえばDX：AXにストアされているダブルワードの値からCX：BXにストアされているダブルワードの値を引く場合は，次のようになります。

```
    .
    .
    .
sub     ax,bx
sbb     dx,cx
    .
    .
    .
```

ADCや**SBB**を使う際は，キャリフラグが前回の加算や減算によって変化していないことを確かめる必要があります。そうしないとキャリフラグにセットされていたキャリまたは借りのステータスは失なわれてしまいます。次の例では，CX：BXとDX：AXは正しく加算されません。

```
    .
    .
    .
add      ax,bx            ; 下位ワードを加える
sub      si,si            ; SIに0をセットする(キャリフラグをクリアする)
adc      dx,cx            ; 上位ワードを加える...
                          ; 前回の加算の後でキャリフラグが壊されているので
                          ; この計算は正しくない.
```

インクリメントとデクリメント

アセンブラプログラムでデータを1だけ大きくしたい場合は，1を加えるのがよいでしょう。これを**インクリメント**と言います。同様にレジスタやメモリ変数から1を引くことを**デクリメント**と言います。1つずつカウントダウンまたはカウントアップするときやメモリのポインタレジスタを大きくするときは，インクリメントとデクリメントで必要な加減算を行なうことができます。

インクリメントやデクリメントはよく使われる操作です。8086は**INC**と**DEC**という命令を持っています。**INC**はレジスタやメモリ変数に**1**を加え，**DEC**はレジスタやメモリ変数から1を引きます。

次の例は10バイトの配列*TempArray*に，0, 1, 2, 3, 4, 5, 6, 7, 8, 9の数値をストアします。

```
        .
        .
        .
      .DATA
TempArray       DB      10 DUP (?)
FillCount       DW      ?
        .
        .
        .
      .CODE
        .
        .
        .
      mov   al,0                    ; TempArrayを初期化
      mov   bx,OFFSET TempArray     ; TempArrayをBXにロード
      mov   [FillCount],10          ; 要素カウンタの初期値をセット
FillTempArrayLoop:
      mov   [bx],al                 ; 現在の要素をTempArrayにストア
      inc   bx                      ; TempArrayの次の要素を指し示す
      inc   al                      ; 値をインクリメントする
      dec   [FillCount]             ; ストアした要素のカウンタをデクリメントする
      jnz   FillTempArrayLoop       ; すべての要素がストアされていなければ次の要素へ
        .
        .
        .
```

結果は同じなのにadd bx, 1のかわりにinc bxを使うのは，**ADD**は3バイト命令ですが**INC**は1バイト命令で実行速度も速いからです。したがって**2つのINC**命令を使う方が，ワードサイズのレジスタに2を加えるよりもコードがコンパクトになります(バイトサイズのレジスタとメモリ変数のインクリメント/デクリメントは2バイト命令ですが，加算命令や減算命令よりはコンパクトです)。

INCと**DEC**はレジスタとメモリ変数の値を1つずつ増加または減少するためのもっとも効率のよい命令です。利用できる場所ではできるだけ使いましょう。

乗算と除算

8086は数種類の整数の乗算と除算ができます。これは8086の特長の1つです。多くのマイクロプロセッサは乗算と除算を直接サポートしていません。ソフトウェアで乗算と除算を行なうのは相当複雑です。

MUL命令は，8ビットまたは16ビットの符号なし要素をかけて16ビットか32ビットの結果を出します。はじめに8ビットどうしの乗算を説明します。

8ビットどうしの乗算の要素の一方はALに，他の要素は，8ビットの汎用レジスタかメモリオペランドに設定しておきます。**MUL**は常に16ビットの結果をAXにストアします。次の例ではALにDHをかけて，その結果の1,000をAXにストアします。

```
	.
	.
	.
	mov	al,25
	mov	dh,40
	mul	dh
	.
	.
	.
```

MULで必要なオペランドは1つだけです。もう1つの要素は常にAL（16ビット同士の乗算ではAX）です。

16ビットどうしの乗算も同様です。1つの要素は必らずAXに設定し，他の要素は16ビットの汎用レジスタかメモリオペランドに設定しておきます。MULは32ビットの結果をDX：AXにストアします。計算結果の下位16ビットをAXに，上位16ビットをDXにストアします。次に例を示します。

```
	.
	.
	.
	mov	ax,1000
	mul	ax
	.
	.
	.
```

この例ではAXに1,000をロードしてAXを自乗します。計算結果の1,000,000はDX：AXにストアされます。

加減算と違って，乗算と除算ではオペランドの符号の有無が問題となります。8ビットと16ビットの符号付き数値の乗算命令に**IMUL**があります。この命令は符号付きの値を扱うということを除いて**MUL**とまったく同じです。次の例ではAXに-20をストアします。

```
	.
	.
	.
	mov	al,-2
	mov	ah,10
	imul	ah
	.
	.
	.
```

8086はある制限のもとで，32ビットの値を16ビットの値でまたは16ビットの値を8ビットの値で割ることができます。16ビットを8ビットで割る場合を最初に説明します。

16ビットを8ビットで割る符号なしの除算の場合，割られる数はAXに設定し，8ビットの除数は8ビットの汎用レジスタかメモリ変数に設定しておきます。**DIV**では8ビットの商は

常にALにストアされ，8ビットの余りはAHにストアされます。次にその例を示します。

```
.
.
.
mov     ax,51
mov     dl,10
div     dl
.
.
.
```

この例では計算結果の5（51/10の商）はALに，1（51/10の余り）はAHにストアされます。

商が8ビットであるため，16ビットを8ビットで割った結果は255以下でなければなりません。もし商が大きすぎると，割り込み0（ゼロ除算の割り込み）が起こります。次の例はゼロ除算の割り込みが起こります（これは0を除数として使ったときにも起こります)。

```
.
.
.
mov     ax,0ffffh
mov     bl,1
div     bl
.
.
.
```

32ビットを16ビットで割る場合，割られる数はDX：AXに，16ビットの除数は16ビットの汎用レジスタやメモリ変数に設定しておきます。DIVでは常に16ビットの商はAXにストアされ，16ビットの余りはDXにストアされます。次の例は計算結果の1000h（10002h/10hの商）はAXに，2（10002h/10hの余り）はDXにストアされます。

```
.
.
.
mov     ax,2
mov     dx,1        ; DX:AXに10002hをロードする
mov     bx,10h
div     bx
.
.
.
```

商は必らず16ビットの値になります。32ビットを16ビットで割った結果は0FFFFhつまり65, 535以下でなければなりません。それ以上だと，ゼロ除算の割り込みが生じます。

乗算と同じく除算の場合も，使われているオペランドが符号付きか符号なしかに注意し

なければなりません。**DIV**は符号なしのオペランドに，**IDIV**は符号付きのオペランドにそれぞれ使います。次の例ではAXに-6を，DXに-67をストアします。

```
    .
    .
    .
    .DATA
RestDivisor    DW   100
    .
    .
    .
    .CODE
    .
    .
    mov     ax,-667
    cwd                            ; DX:AXに-667をセット
    idiv    [TestDivisor]
    .
    .
    .
```

符号の変更

最後に**NEG**命令について説明します。これは汎用レジスタやメモリ変数の値の符号を反転させる命令です。次の例ではAXが-1に，BXが1になります。

```
 .
 .
 .
 mov     ax,1                ; AXに1をセット
 neg     ax                  ; AXの符号を反転，AXは-1になる
 mov     bx,ax               ; AXをBXにコピー
 neg     bx                  ; BXの符号を反転，BXは1になる
 .
 .
 .
```

論理演算

Turbo Assemblerは，**AND, OR, XOR, NOT**などの論理演算をすべてサポートしています。これらの命令はバイトやワードの中の個々のビット操作やブール代数を行なうときに非常に便利です。

2つのソースビットに対する論理演算の結果は表4.1のようになります。論理命令はソース

オペランドに対応するビットの操作を行ないます。次の例はAXのビット0とDXのビット0のANDをとり，結果をAXのビット0にストアします。ビット1～ 15も同様に処理します。

```
and     ax,dx
```

表4.1　8086の論理演算命令AND, OR, XORの処理

ソースビットA	ソースビットB	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

AND命令は2つのオペランドの対応するビットが両方とも1の場合のみ，デスティネーションオペランドの対応ビットを1にセットします。**AND**は特定のビットをマスクしたり，特定のビットを強制的に0にするために使います。次の例はColor/Graphic Adapter(CGA)のステータスバイトのビット0以外をマスクします。

```
.
.
.
mov     dx,3dah
in      al,dx
and     al,1
.
.
.
```

この例では，スノーノイズを発生せずにCGAのディスプレイメモリを更新する場合はALは1のままで，そうでなければALは0になります。

OR命令は2つのオペランドを表4.1に示す規則にしたがって演算します。対応するソースビットのどちらかが1であれば，デスティネーションオペランドに1をセットします。**OR**は特定のビットを強制的に1にします。次の例では，BIOSの装置フラグワードのビット4とビット5をともに1にします。これによってBIOSはモノクロディスプレイアダプタをサポートします。

```
        .
        .
        .
        mov     ax,40h
        mov     ds,ax
        mov     bx,10h
        or      WORD PTR [bx],0030h
        .
        .
        .
```

XOR命令は2つのオペランドを表4.1に示す規則にしたがって演算します。対応するソースビットの一方が0で他方が1の場合のみ，デスティネーションオペランドに1をセットします。これによってバイト中の特定のビットを反転できます。次の例ではALに10100101bつまりA5hをセットします。

```
        .
        .
        .
        mov     al,01010101b
        xor     al,11110000b
        .
        .
        .
```

ここのキーポイントは，**AL**と11110000bつまり0F0hの**排他的論理和**をとれば，0F0hの1のビットは**AL**の対応ビットを反転し，0F0hの0のビットは**AL**の対応ビットをそのままにしておきます。その結果**AL**の上位4ビットはすべて変化し，下位の4ビットはそのままの値になります。

ところで**XOR**はレジスタを0にするのに便利な方法です。次の例はAXを0にセットします。

```
        xor     ax,ax
```

NOTはオペランドの各ビットを反転します。ソースオペランドに0FFhを指定して**XOR**を実行したのと同じです。次に例を示します。

```
        .
        .
        .
        mov     bl,10110001b
        not     bl                  ; BLを01001110bに反転
        xor     bl,0ffh             ; BLを10110001bに戻す
        .
        .
        .
```

シフトとローテーション

8086にはレジスタやメモリ変数のビットを左右にシフトする多くの命令が備わっています。一番簡単な方法は論理シフトです。

SHL（左シフト，**SAL**とも呼ばれます）は，デスティネーションオペランドの各ビットを左へ，つまり最上位ビットの方へシフトします。図4.8はALにストアしてある10010110bという値（96hつまり10進数の150）が**SHL AL, 1**によって左にシフトされるようすを示したものです。この際キャリフラグは1にセットされます。

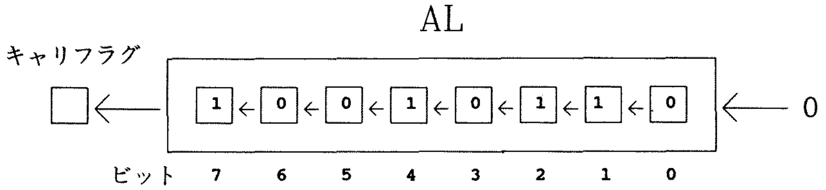


図4.8　左シフトの例

最上位ビットは，オペランドの外にシフトされ，キャリフラグに入ります。最下位ビットには0が入ります。

左シフトはどのようなときに使うのでしょう。**SHL**のもっとも一般的な使い方は高速な2の乗数による乗算です。**SHL**を1回行なうとオペランドに2をかけたことになります。次の例ではDXを16倍します。

```
.
.
.
shl  dx,1       ; DX = 2
shl  dx,1       ; DX = 4
shl  dx,1       ; DX = 8
shl  dx,1       ; DX = 16
.
.
.
```

シフトによる乗算は**MUL**命令を使うよりもずっと高速です。

上の例では，**SHL**の第2オペランドが*1*になっていることに注意してください。これはDXを1ビット左にシフトすることを示します。8086は1以外のシフトはサポートしていませんが，シフトのカウンタとしてCLを使うことができます。次の例では前の例と同様にDXを16

倍します。

```
.
.
.
mov     cl,4
shl     dx,cl
.
.
.
```

左シフトがあるなら，右シフトもあるのが自然です。8086には2種類の右シフト命令があります。**SHR**（shift right）は**SHL**によく似ています。オペランドの中のビットを右に1ビットまたはCLビット分シフトします。最下位ビットはキャリフラグにシフトされ，最上位ビットに0が入ります。**SHR**は2で割る符号なしの除算を高速で実行する方法です。

SAR（arithmetic shift right）は**SHR**によく似ています。ただし，**SAR**ではオペランドの最上位ビットは右隣りにシフトされ，またそれ自身に返ってくるという点が違います。図4.9は，ALの10010110b（96hつまり符号付き10進数の-106）という値が**SAR AL, 1**という命令によって右シフトするようすを示します。その結果，値は11001011b（0CBhつまり符号付き10進数の-53）となり，再びALにストアされます。キャリフラグは0になります。

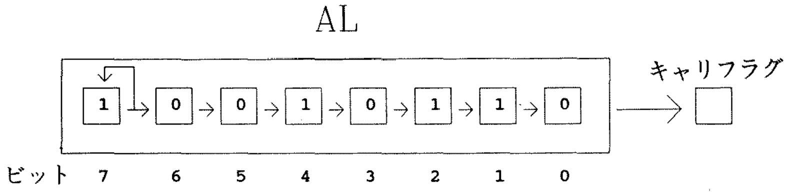


図4.9　SAR（算術的右シフト）の例

この命令はオペランドの符号を保持するという特性を持っています。したがって**SAR**は符号付きの2の除算をするときに便利です。次の例では結果-2をBXにストアします。

```
.
.
.
mov     bx,-4
sar     bx,1
.
.
.
```

ローテーション命令は**ROR, ROL, RCR, RCL**の4種類です。**ROR**は**SHR**に似ていますが，最下位ビットがキャリフラグに入るのと同時に最上位ビットにも入るという点が違っています。図4.10はALの10010110b（96hつまり10進数の150）という値が，**ROR AL, 1**に

よって右にローテーションするようすを示します。結果は01001011b(04Bhつまり10進数の75）となり，再びALにストされます。キャリフラグは0になります。

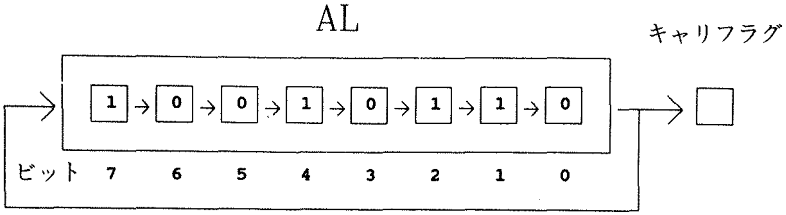


図4.10　ROR（右ローテーション）の例

ROLは**ROR**の逆の動きをします。オペランドを左側へ循環シフトし，最上位ビットを最下位ビットに入れます。**ROR**と**ROL**はバイトやワードのビットの確認に役立ちます。次の例では，3-0ビットが15-12に，7-4ビットが3-0に転送され，その他のビットも同じように転送されます。SIは149Fhになります。

```
.
.
.
mov     si,49F1h
mov     cl,4
shl     si,cl
.
.
.
```

RCRと**RCL**は少し違います。**RCR**は右シフトに似ていますが，最上位ビットにキャリフラグの値が入ります。図4.11は，キャリフラグが1のときALの10010110b(96hつまり10進数の150)を**ROR AL, 1**によって右にシフトするようすを示します。結果は，ALは11001011b（0CBhつまり10進数の203）に，キャリフラグは0になります。

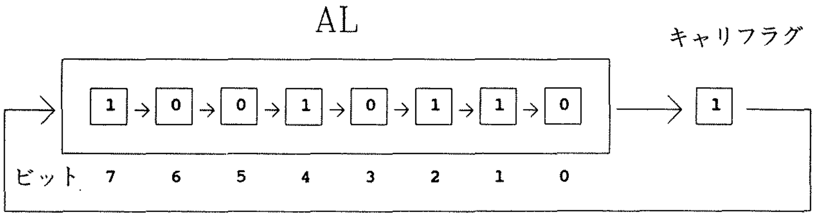


図 4.11　RCR（キャリを伴う右ローテーション）の例

同様に**RCL**は最下位ビットにキャリフラグの値が入る左シフトです。**RCR**と**RCL**は複数のワードオペランドをシフトする場合に便利です。次の例ではDX AXのダブルワードの値

に4をかけたことになります。

```
  .
  .
  .
  shl   ax,1          ; AXのビット15をキャリにシフト
  rcl   dx,1          ; キャリをDXのビット0にシフト
  shl   ax,1          ; AXのビット15をキャリにシフト
  rcl   dx,1          ; キャリをDXのビット0にシフト
  .
  .
  .
```

シフト命令と同様, ローテーション命令はオペランドを1ビットずつまたはCLで指定したビット数だけシフトすることができます。

ループとジャンプ

ここまでは順番に実行する命令について説明してきました。それぞれの命令は，前のアドレスの命令の直後に実行されます。次のようなプログラムでは，**ADD**は**MOV**の直後に実行され，**PUSH**はその後に実行されます。

```
.
.
.
mov     ax,[BaseCount]
add     ax,4
.
.
.
push    ax
.
.
.
```

もし8086がこのような方法でしかプログラムを実行できないのなら，実に役に立たないコンピュータといえます。役に立つコンピュータの基本的な性能は，メモリ上の任意の命令にジャンプするための命令を持っていることです。同様に大切なことは，ステータスや処理の結果によって分岐条件を変える機能です。当然8086はこれら両方の分岐命令を備えています。さらに命令ブロックを簡単に繰り返し実行する特別な分岐命令も備えています。

無条件ジャンプ

8086の基本的な分岐命令は**JMP**命令です。この命令は分岐先のラベルにある命令を実行するように指示する命令です。次の例を実行するとAXは3になりますが，*AddOneToAX*というラベルの後の**INC**と**JMP**は実行されません。

```
          .
          .
          .
          mov     ax,1
          jmp     AddTwoToAX
AddOneToAX:
          inc     ax
          jmp     AXIsSet
AddTwoToAX:
          inc     ax
AXIsSet:
          .
          .
          .
```

この例では，

```
jmp  AddTwoToAX
```

という命令によって命令ポインタIPにラベル*AddTwoToAX*のオフセットをセットし，次に

```
add  ax,2
```

を実行します。

JMP命令に演算子**SHORT**を使うことがたまにあります。JMPでは目的のラベルとして通常16ビットの変位を使います。**SHORT**は16ビットのかわりに8ビットの変位を使うことを指示します。したがって1つの**JMP**命令につき1バイトを節約します。したがって次の例の方が前の例より2バイトだけ短かくなります。

```
          .
          .
          .
          mov     ax,1
          jmp     SHORT AddTwoToAX
AddOneToAX:
          inc     ax
          jmp     SHRORT AXIsSet
AddTwoToAX:
          inc     ax
AXIsSet:
          .
          .
          .
```

SHORTの欠点は，**JMP**命令から128バイト以内にしかジャンプできないという点です。したがってshortジャンプでは目的のラベルに到達できない旨のメッセージが出されることもあります。**SHORT**を使えるのは前方ジャンプだけです。後方ジャンプでは，shortジャンプで目的のところへ分岐できる場合は自動的にshortジャンプを行ない，そうでない場合

は自動的にlongジャンプを行ないます。

JMPでは，CSとIPをセットして，別のコードセグメントへfarジャンプすることもできます。次にfarジャンプの例を示します。

```
        .
        .
        .
CSeg1   SEGMENT
        ASSUME  CS:Cseg1
        .
        .
        .
FarTarget       LABEL    FAR
        .
        .
        .
Cseg1   ENDS
        .
        .
        .
Cseg2   SEGMENT
        ASSUME  CS:Cseg2
        .
        .
        .
        jmp     FarTarget                   ; farジャンプ
        .
        .
        .
Cseg2   ENDS
        .
        .
        .
```

必要なら**FAR PTR**演算子を使って，ラベルをfarラベルとして扱うこともできます。次の例では**JMP**命令と同じコードセグメント内の*NearLabel*へfarジャンプします。

```
        .
        .
        .
        jmp     FAR PTR NearLabel
        nop
NearLabel:
        .
        .
        .
```

さらに，レジスタやメモリ変数にストアされているアドレスにジャンプ(間接ジャンプ)することもできます。次の2つの例はどちらも*TestLabel*に分岐します。

```
        .
        .
        .
        mov     ax,OFFSET TestLabel
        jmp     ax
        .
        .
        .
TestLabel:
        .
        .
        .
```

および

```
        .
        .
        .
        .DATA
JumpTarget      DW    TestLabel
        .
        .
        .
        .CODE
        .
        .
        .
        jmp    [JumpTarget]
        .
        .
        .
TestLabel:
        .
        .
        .
```

条件ジャンプ

前の節で説明したジャンプは，役に立つプログラムを書く上で必要なもののごく一部に過ぎません。プログラムで本当に必要なのは，条件判断を伴うジャンプです。次に条件ジャンプについて説明します。

条件ジャンプ命令は，フラグレジスタの状態によって目的のラベルに分岐したりしなかったりします。次に例を示します。

```
        .
        .
        .
        mov     ah,1                    ; キーボード入力のDOSファンクション
        int     21h                     ; 次のキーを押す
        cmp     al,'A'                  ; 入力が大文字の'A'かどうかの判断
        je      AWasTyped               ; 大文字'A'なら，特別な処理
        mov     [TempByte],al           ; そうでなければ，文字を保存
        .
        .
        .
AWasTyped:
        push    ax                      ; 文字をスタックにセーブ
        .
        .
        .
```

この例は，DOSファンクションコールを使って押されたキーの値を得ます。次に**CMP**命令を使って，タイプされた文字を*'A'*と比較します。**CMP**命令は**SUB**命令のようにはオペランドに影響を与えません。**CMP**は2つのオペランドを比較するだけでオペランドの内容は変えません。しかし**CMP**は**SUB**と同様にフラグをセットします。したがってこの例では，**AL**の内容が*'A'*ならばゼロフラグに1がセットされます。

この例の中心の**JE**命令は，ゼロフラグが1のときだけ分岐する条件ジャンプ命令です。ゼロフラグが1でないときは**JE**の直後の命令（この例では**MOV**命令）を実行します。したがって*'A'*のキーが押された場合だけゼロフラグに1がセットされ，この場合だけラベル*AWasTyped*の**PUSH**命令に分岐します。

8086は多様な条件ジャンプ命令を持っています。フラグの組み合わせによる非常に多くの種類のジャンプがあります。ゼロフラグ，キャリフラグ，サインフラグ，パリティフラグ，オーバフロー フラグの状態や，符号付き数値の演算結果を示すフラグの状態の組み合わせによって条件ジャンプが行なわれます。

表4.2に条件ジャンプ命令をまとめておきます。

表4.2　条件ジャンプ命令

命令	ジャンプする条件	チェックされるフラグ
JB/JNAE	未満/以上でない	CF=1
JAE/JNB	以上/未満でない	CF=0　　符号なしデータのとき
JBE/JNA	以下/大きくない	CF=0　または　ZF=1
JA/JNBE	大きい/以下でない	CF=0　かつ　ZF=0
JE/JZ	等しい	ZF=1　　付け加える
JNE/JNZ	等しくない	ZF=0
JL/JNGE	未満/以上でない	SF≠OF
JGE/JNL	以上/未満でない	SF=OF　　符号なしデータのとき
JLE/JNG	以下/大きくない	ZF=1　または　SF=OF
JG/JNLE	大きい/以下でない	FZ=0　かつ　SF=OF
JP/JPE	パリティが1/パリティフラグが1	PF=1　　付け加える
JNP/JPO	パリティが0/パリティフラグが0	PF=0
JS	サインフラグが1	SF=1
JNS	サインフラグが0	SF=0
JC	キャリフラグが1	CF=1
JNC	キャリフラグが0	CF=0
JO	オーバフロー フラグが1	OF=1
JNO	オーバフロー フラグが0	OF=0

CF=キャリフラグ，SF=サインフラグ，OF=オーバフローフラグ，
ZF=ゼロフラグ，PF=パリティフラグ

同義語や一般的な条件ジャンプ命令については第5章で説明します。そこでは8086の各命令でフラグレジスタを変化する方法についても詳しく説明しています。

条件ジャンプには柔軟性がありますが厳しい制限もあります。条件ジャンプは必らずshortジャンプでなければなりません。つまり，条件ジャンプ命令のジャンプ先ラベルは，その命令から128バイト以内になくてはなりません。したがって次の例は正しくありません。

```
        .
        .
        .
        .
JumpTarget:
        .
        .
        .
        DB      1000 DUP (?)
        .
        .
        .
        dec     ax
        jnz     JumpTarget
        .
        .
        .
```

この例では，ジャンプ先の*JampTarget*がJNZ命令から1,000バイト以上離れています。正しくは次のようにします。

```
        .
        .
        .
        .
JumpTarget:
        .
        .
        .
        DB      1000 DUP (?)
        .
        .
        .
        dec     ax
        jz      SkipJump
        jmp     JumpTarget
SkipJump:
        .
        .
        .
```

この例では，条件ジャンプを使って無条件のlongジャンプを行なうか否かを判断しています。

ループ処理

条件ジャンプによるプログラムの制御構造の1つにループがあります。ループは条件ジャンプで終了する命令ブロックですが，ブロックは終了条件に達するまで繰り返し実行され

ます。ループはCの**for**文や**while**文，Pascalの**repeat**文，BASICのfor文などでよく知られています。

ループは，配列処理，一定条件に達するまでのI/Oポートの状態のテスト，メモリの一括クリア，キーボードからの文字列の読み取り，文字列のスクリーンへの表示などさまざまな処理に使われます。このように，ループは特定の動作を繰り返すときの基本的な方法です。ループはきわめて頻繁に使われます。8086のループ処理命令は，**LOOP, LOOPE, LOOPNE, JCXZ**です。

はじめに**LOOP**について説明しましょう。*TestString*という文字列にある17の文字を表示するプログラムを例にとります。

```
        .
        .
        .
        .DATA
TestString      DB    'This is a test....'
        .
        .
        .
        .CODE
        .
        .
        .
        mov     dl[bx]                  ; 次の文字を取り込む
        inc     bx                      ; 次の文字を指し示す
        moz     ah,2                    ; DOSのファンクションで画面に表示
        int     21h                     ; 文字を表示するファンクションコール
        dec     cx                      ; 文字列の長さを1だけ減らす
        jnz     PrintStringLoop
                                        ; 次の文字があれば繰り返す
        .
        .
        .
```

この例ではもっとよい方法があります。すでに述べたように，このようなループプログラムにはCXが役に立ちます。つまり

```
dec     cx
jnz     PrintStringLoop
```

のかわりに

```
loop    PrintStringLoop
```

を使うことができます。この命令を使う方が高速でかつ1バイト少なくなります。カウンタがゼロになるまで繰り返すループが必要なときはいつでも，カウントをCXにセットして**LOOP**命令を使います。

単純なカウントダウンではなく終了条件がある場合は，**LOOPE**と**LOOPNE**を使います。

LOOPEは**LOOP**と同じことをしますが，**LOOPE**はCXがゼロになったときまたはゼロフラグが1にセットされたときにループを抜けます（分岐しなくなります）（計算結果がゼロ，または比較した2つのオペランドが等しくない場合は，ゼロフラグは1にセットされます）。同様に**LOOPNE**はCXがゼロになるか，ゼロフラグが0になったときにループを終了します。

リターンキーが押されるか，または128文字を読み取るまでキーの読み込みを繰り返す例を考えます。次の例では**LOOPNE**を使っています。

```
        .
        .
        .
        .DATA
KeyBuffer      DB      128 DUP (?)
        .
        .
        .
        .CODE
        .
        .
        .
        mov     cx,128
        mov     bx,OFFSET KeyBuffer
KeyLoop:
        mov     ah,1              ; キーボード入力のDOSファンクション番号
        int     21h               ; 次のキーを読み取る
        mov     [bx],al           ; キーの値をストア
        inc     bx                ; ポインタを次のキーにセット
        cmp     al,0dh            ; リターンキーかどうかチェック
        loopne KeyLoop            ; リターンキーでない，あるいはキーの最大数に
                                  ; 達していなければ，次のキーを読み取る
        .
        .
        .
```

JEと**JZ**が同じであったように，**LOOPE**と**LOOPZ**, **LOOPNE**と**LOOPNZ**はそれぞれ同じです。

ループに関するもう1つの命令に**JCXZ**があります。**JCXZ**の分岐は，CXが0のときのみ実行されます。これは，ループに入る前にCXをテストするのに便利です。次の例では，BXは0にセットすべき連続するバイトブロックを指し，CXにはブロックの長さが入っています。**JCXZ**を実行してCXがゼロならループ全体をスキップします。

```
        .
        .
        .
        jcxz    SkipLoop                 ; もしCXが0なら何もしない
ClearLoop:
        mov      BYTE PTR [si],0
                                         ; 次のバイトを0にセット
        inc     si                       ; クリアする次のバイトを指し示す
        loop    ClearLoop                ; 次の文字があればクリア
SkipLoop:
        .
        .
        .
```

CXが0のときにスキップする理由は，CXが0のときに**LOOP**を実行するとCXはデクリメントされて0FFFFhになり，**LOOP**命令の目的のラベルへ分岐してしまうからです。その場合はループはさらに65,536回実行されます。この例では，CXが0になったら，メモリに0をストアする処理を終了しなければなりません。65,536バイトのメモリに0をストアするのが目的ではありません。**JCXZ**は，このような場合に効率よくCXをテストします。

ループ命令については2つの注意点があります。第1は，条件ジャンプと同様にループ命令はその命令の前後128バイトの範囲内のラベルにしか分岐できないことです。128バイトより大きいループには前の「条件ジャンプ」の節で説明した「無条件ジャンプと条件ジャンプの組み合わせ」のテクニックを使います。第2に，すべてのループ命令はフラグには影響を与えません。したがって

```
loop    LoopTop
```

と

```
dec     cx
jnz     LoopTop
```

はまったく同じではありません。

LOOP命令ではどのフラグも変わりませんが，**DEC**命令ではオーバフローフラグ，サインフラグ，ゼロフラグ，補助キャリフラグ，キャリフラグ，パリティフラグなどが変化します。同様に**DEC**命令は次の例とも違います。

```
sub    cx,1
jnz    LoopTop
```

DEC命令ではキャリフラグは変わりませんが，**SUB**命令では変わります。このように，フラグの変化は命令によって少しずつ違いがあります。大切なのは，アセンブリ言語のプログラミングでは命令セットをしっかり理解することです。

サブルーチン

ここまでは，連続したコードから成るプログラムについて説明してきました。各プログラムは先頭から開始して各命令を順番に（ループや判断命令でときどき分岐します）実行し，コードの終わりで終了します。小さいプログラムではこれで充分ですが，大きいプログラムでは**サブルーチン**といわれるプログラミング構造が必要になってきます。

ほとんどのプログラマは，高級言語の経験からサブルーチンには馴染みがあることと思います。Cではサブルーチンは関数と呼ばれ，PascalやBasicでは手続き（プロシージャ）または関数と呼ばれます。サブルーチンもプロシージャも関数も同じことです。明確に定義された入力を受け付け，特定の処理を行ない，必要に応じて値を返す独立したプログラムです。

サブルーチンを使うと，プログラムをモジュールに分割して作ることができます。また細かい処理を隠して，プログラム全体の流れに焦点を当てることができます。1つのサブルーチンはプログラムの中の複数の箇所から呼び出すことができ，しかも値によって異なる処理を行なうため，プログラムをコンパクトにすることができます（アセンブラ，C，Pascal，その他のどんな言語で書かれていても）長いプログラムを作る際は，サブルーチンを使うことは整然としたメンテナンスしやすいコードを作るための基本となります。

サブルーチンの動作

サブルーチンの基本的な処理を図4.12に示します。サブルーチンはCALL命令によって呼び出されます。CALLは次の命令のアドレスをスタックにプッシュし，目的のサブルーチンのアドレスをIPにセットします。そしてサブルーチンに分岐し，サブルーチンを通常のプログラムと同じように実行します。サブルーチンから他のサブルーチンをコールすることが可能です（これはよく行なわれます）。正しく設計されたサブルーチンは，自分自身さえもコールすることが可能です。これは**リカージョン（再帰）**として知られています。

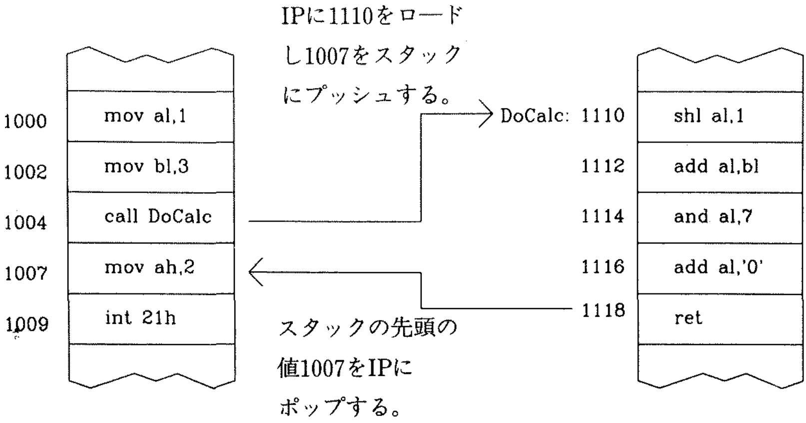


図4.12　サブルーチンの動作

サブルーチンの処理が終わると，最後に**RET**命令を実行します。RET命令は元の**CALL**命令でプッシュしたアドレスをIPにポップし，CALLの直後の命令から元のルーチンの処理を再開します。　次に示す例は，*PrintString*というサブルーチンを使って次のような3つの文字列を表示します。

```
Hello, world!
Hello, solar system!
Hello, universe!
```

プログラムは次のようになります。

```
        DOSSEG
        .MODEL   SMALL
        .STACK   200h
        .DATA
WorldMessage     DB     'Hello, world!',0dh,0ah,0
SolarMessage     DB     'Hello, solar system!',0dh,0ah,0
UniverseMessage  DB     'Hello, universe!',0dh,0ah,0
        .CODE
ProgramStart     PROC    NEAR
        mov     ax,@Data
        mov     ds,ax
        mov     bx,OFFSET WorldMessage
        call    PrintString                ; Hello, world!と表示
        mov     bx,OFFSET SolarMessage
        call    PrintString                ; Hello, soler system!と表示
        mov     bx,OFFSET UniverseMessage
        call    PrintString                ; Hello,universe!と表示
        mov     ah,4ch                     ; 終了のDOSファンクション番号
        int     21h                        ; ファンクションコール実行
ProgramStart    ENDP
;
; ヌルを終端とする文字列を画面に表示するサブルーチン
;
```

```
; 入力:
;       DS:BX - 表示する文字列を指すポインタ
;
; 破壊されるレジスタ  AX,BX
;
PrintString    PROC    NEAR
PrintStringLoop:
        mov     dl,[bx]                      ; 文字列の次の文字を取り込む
        and     dl,dl                        ; 文字の値のゼロテスト
        jz      EndPrintString               ; ゼロなら文字列は終わり
        inc     bx                           ; 次の文字を取得
        mov     ah,2                         ; 画面表示のDOSファンクション
        int     21h                          ; ファンクションコール実行
        jmp     PrintStringLoop              ; 次の文字があれば表示
EndprintString:
        ret                                  ; 呼び出したプログラムに戻る
        PrintString
        ENDP
        END     ProgramStart
```

この例では，注意しなくてはならないことが2つあります。第1は，*PrintString*は特定の文字列を表示するのではなく，呼び出し側プログラムのBXが指し示す文字列を表示するということです。第2は，*PrintString*の中で**PROC**と**ENDP**という2つの新しい命令を使っている点です。

PROCはプロシージャのはじめに使います。**PROC**の付いたラベル(ここでは*PrintString*)はプロシージャの名前で，次のように書くのと同じです。

```
PrintString   LABEL   PROC
```

PROCには別の役割もあります。つまりプロシージャの**RET**をnear型にするかfar型にするかを指定します。

この意味についてもう少し詳しく説明します。nearラベルに分岐するときは，IPにアドレスをセットし，farラベルに分岐するときはCSとIPにアドレスをセットすることに注意してください。**CALL**命令がfarラベルを参照すれば，ジャンプと同様にCSとIPにアドレスがセットされます。

farコールのときはCSとIPをともにプッシュしないと，**RET**命令によって元のプログラムに戻るための必要な情報が得られません。farコール時にCSとIPをセットしてIPだけをプッシュすると，リターン時にはIPにのみ（スタックの先頭から）アドレスがセットされます。その結果，CS：IPは呼び出し側ルーチンのIPと呼び出されたルーチンのCSの組合わせになってしまい，わけの解らないことになってしまいます。

実際にfarラベルをコールすると，CSとIPがともにプッシュされます。サブルーチンのリターンの型(farリターンかnearリターン)を指示する方法は2つあります。1つは**RETN**(nearリターン）命令または**RETF**（farリターン）命令でリターンの型を指定する方法です。もう1つは，**PROC**疑似命令と**ENDP**疑似命令を使って指定する方法です。

サブルーチンの始まりを**PROC**で示すのと同様に，サブルーチンの終わりは**ENDP**命令で示します。**ENDP**には，**PROC**がサブルーチンのはじめに付けたラベルと同じラベルを指定してサブルーチンの終わりを示します。次の例は，*TestSub*というサブルーチンのはじまりと終わりを示しています。

```
          .
          .
          .
TestSub        PROC      NEAR
          .
          .
          .
TestSub        ENDP
          .
          .
          .
```

PROCと**ENDP**は，実際には何のコードも生成しません。これらは疑似命令で，このサブルーチンの**RET**の型を指定します。

PROC疑似命令のオペランドが**NEAR**の場合は，**PROC**疑似命令とそれに対応する**ENDP**疑似命令の間にある**RET**命令は，全部nearリターンとしてアセンブルされます。一方**PROC**疑似命令のオペランドが**FAR**の場合は，このプロシージャの中の**RET**命令は全部farリターンとしてアセンブルされます。したがって，たとえば*TestSub*の**RET**命令を全部**far**に変えたい場合は，**PROC**疑似命令を次のように変えればよいのです。

```
TestSub   PROC   FAR
```

一般に，nearが使えるときはできるだけnearを使います。farコールはnearコールよりもプログラムのサイズが大きくなり，実行速度が遅くなるからです。farリターンもnearリターンより実行速度は遅くなります。しかし64Kバイト以上のプログラムではfarサブルーチンが必要になります。

簡略セグメント疑似命令を使う場合は，次の例のように**PROC**疑似命令にはオペランドを付けない方がよいでしょう。

```
TestSub    PROC
```

Turbo Assemblerはこのような疑似命令を見つけると，**.MODEL**命令で選択したメモリモデル（デフォルトはスモール）に基づいてnearプロシージャまたはfarプロシージャを自動的に作ります。タイニィモデル，スモールモデル，コンパクトモデルのプログラムではnearコールになり，ミディアムモデル，ラージモデル，ヒュージモデルのプログラムではfarコールになります。次の例では，*TestSub*はnearコールで呼び出すことができます。

```
        .
        .
        .
        .
        .MODEL SMALL
        .
        .
        .
TestSub PROC
        .
        .
        .
```

一方次の例では，*TestSub*はfarコールで呼び出すことができます。

```
        .
        .
        .
        .
        .MODEL LARGE
        .
        .
        .
TestSub PROC
        .
        .
        .
```

パラメータの受け渡し

多くの場合，サブルーチンには呼び出し側プログラムから情報が渡されます。たとえば前の節で例示したプログラムは，BXレジスタをポインタとして*PrintString*サブルーチンに渡しています。この働きは**パラメータの受け渡し**と呼ばれます。パラメータはサブルーチンが行なう処理を指示します。

パラメータの受け渡しには，レジスタを使う方法とスタックを使う方法の2つの方法があります。レジスタによる受け渡しは純粋なアセンブラプログラムが使う方法です。これに対してスタックによる受け渡しは，PascalやCなどのほとんどの高級言語のプログラムからアセンブラサブルーチンをコールする場合に使われます。

レジスタによるパラメータの受け渡しは簡潔です。パラメータの値を適当なレジスタに入れてサブルーチンをコールします。それぞれのサブルーチンは独自のパラメータ形式をもっています。混乱を避けるためには，規定を作ってそれを厳守することです。たとえば第1ポインタパラメータはBXに，第2ポインタパラメータはSIに，というルールを作ってお

くとよいでしょう。レジスタによる受け渡しを使う場合は，パラメータの内容とそれがどのレジスタに設定されるのかを，各サブルーチンの先頭に正しくコメントしておくことが大切です。

スタック経由のパラメータの受け渡しは少し複雑です。この場合は，得意な高級言語のプログラムとアセンブラサブルーチンのリンク規定にしたがってください。第6章～9章でTurbo C, Turbo Pascal, Turbo Basic, Turbo Prologにおけるパラメータ受け渡しの規定について詳しく説明し，アセンブラのプログラム例も示します。

リターン値

多くの場合，サブルーチンは呼び出し側プログラムに値を返します。高級言語から呼び出されるアセンブラのサブルーチンは，高級言語のリターン値に関する規定に**従わなければなりません**。たとえばCから呼び出す関数はAXに8ビットか16ビットの値(**chars, int, near**ポインタ)を，DX：AXに32ビットの値（**longs**と**far**ポインタ）を返さなければなりません。このマニュアルの第6章～9章でTurbo C, Turbo Pascal, Turbo Basic, Turbo Prologにおけるリターン値の規定について詳しく述べます。

純粋なアセンブラプログラムでは，どのように値を返すかはまったく自由です。任意のレジスタに入れて返すことができます。またフラグレジスタ，キャリフラグ，ゼロフラグに設定してステータス情報を返すこともできます。しかし繰り返しますが，いくつかの規定を作ってそれに従うのが最良です。たとえば8ビットの値はALに，16ビットの値はAXに入れて返すと決めると便利です。そうすれば，不用意にAXに大切なデータを入れない習慣がつきます。

アセンブラでサブルーチンが返す値を使うときの主な問題点は，値を返す処理の過程で呼び出し側ルーチンの重要な情報をサブルーチンが破壊する可能性があることです。アセンブラでは，サブルーチンがたとえばSIに値を返すことを忘れて，そのサブルーチンをコールすることがあります(サブルーチンはSIの値を簡単に変えてしまいます)。そのようなときに発生するバグは，非常に見つけにくいものです。

したがって，サブルーチンがレジスタに入れて返す値は，できるだけ1つに絞ったほうがよいでしょう。他の値はCやPascalのように，渡されたポインタが指し示すメモリを介して返すのがよいでしょう。

レジスタの退避

サブルーチンコールにおけるレジスタの退避は，通常アセンブラプログラミングの主要な問題点です。最近の高級言語では，呼び出しプログラムが特別に作られていない限り，サブルーチンが呼び出しプログラムの変数を変えることはできません。ところがアセンブラプログラムでは，サブルーチンが使うレジスタに呼び出し側プログラムの変数が格納されることがよくあります。たとえば，サブルーチンコールの前でセットしたレジスタの値をサブルーチンが変更してしまい，サブルーチンコールの後でそのことを知らずにそのレジスタの値を使うとバグが発生します。

この問題の解決方法の1つは，それぞれのサブルーチンの入口でサブルーチン内で使うすべてのレジスタを常にプッシュし，出口でポップしてレジスタを復帰する方法です。ただしこの方法は時間を無駄に使い，ステップ数も増えてしまいます。もう1つの方法は，サブルーチンがレジスタを破壊することを前提にプログラムを作ることです。したがって処理に関連するすべてのレジスタは，呼び出し側プログラムで面倒を見なければなりません。しかしアセンブラを使う最大のメリットは，レジスタを自由に効率よく使えることですから，この方法もあまり有益ではありません。

アセンブリ言語では，プログラムの処理速度とコーディングのしやすさは両立しません。アセンブラを使うのなら，高速でコンパクトなプログラムを書かなくてはなりません。そのためには，レジスタの退避についてよく理解し，サブルーチンコールによってレジスタが破壊されないように配慮しなくてはなりません。最良の方法は，それぞれのサブルーチンが破壊するレジスタをコメントとして記述しておき，サブルーチンを使う際にそのコメントを参照することです。

レジスタの退避に注意しながら，レジスタをできるだけ効率的に使うことがアセンブラプログラミングでは重要です。高級言語ではこれらの処理を言語のコンパイラが行ないます。繰り返しますが，高級言語ではアセンブラプログラムのように高速でコンパクトなプログラムを作ることはできません。

アセンブラのサンプルプログラム

前章とこの章で説明してきたことを実際に役立つプログラムに応用してみましょう。WCOUNT.ASMという名のこのサンプルプログラムは，ファイル内の単語数をカウントしてそれを画面に表示します。

```
; ファイル内の単語数をカウントするプログラム. 単語はホワイトスペースで区切ら
; れていること. ホワイトスペースは, スペース, タブ, 復帰, 改行のどれか.
;
; 使用方法: wcount <filename.ext>
;
     DOSSEG                                  ; 標準セグメントオーダ
     .MODEL     SMALL                        ; コードとデータはそれぞれ64K以内
     .STACK     200h                         ; 512バイトのスタック
     .DATA
COUNT          DW   0                        ; 単語カウンタ
InWhitespace DB    ?                        ; 読んだ文字がホワイトスペースなら
                                             ; 1にセット
TempChar       DB?                           ; GetNextCharacterが使用する一時的
                                             ; データ保管場所
Result         DB   'Word count: ',5 DUP(?)
                                             ; 単語数表示用の文字列
CountInsertEnd LABEL BYTE                    ; カウント値をストアする領域の最後を
                                             ; 見つけるために使用
               DB   0dh,0ah,'$'
                                             ; DOSファンクション#9を使うときは,
                                             ; 文字列の終端に$を付ける

     .CODE
ProgramStart
        mov     ax,@data
        mov     ds,ax                        ; DSにDATAセグメントをセット
        mov     [InWhitespac],1              ; 初期値をホワイトスペースとする.
                                     ; 次に読み込んだホワイトスペース以外の
                                             ; 文字が単語の先頭となる.
CountLoop:
        call    GetNextCharacter             ; 次の文字を取り込む
        jz      CountDone                    ; ...文字があればジャンプ
        call    IsCharacterWhitespace        ; 文字はホワイトスペースか?
        jz      Whitespace                   ; Yes
        cmp     [InWhitespace],0             ; 文字はホワイトスペースではない
                                             ; 現在の文字はホワイトスペースか?
        jz      CountLoop                    ; 現在の文字も次の文字もホワイト
                                             ; スペースではない.
        inc     [Count]                      ; 現在の文字がホワイトスペースで,
                                             ; 次の文字はホワイトスペースであれば
                                             ; 新しい単語の先頭を検出.
        mov     [InWhitespace],0             ; 現在文字がホワイトスペースではない
                                             ; ことを示す
        jmp     CountLoop                    ; 次の文字の処理へ
```

```
IsWhitespace:
        mov '  [InWhitespace],1
                                        ; 現在文字がホワイトスペースであること
                                        ; を示す
        jmp     CountLoop               ; 次の文字の処理へ
;
; カウント値を表示
;
CountDone:
        mov     ax,[Count]              ; 数値を文字列に変換
        mov     bx,OFFSET CountInsertEnd-1
                                        ; 数値をストアするために,
                                        ; 文字列の終わりを指し示す
        mov     cx,5                    ; 変換する桁数
        call    ConvertNumberToString   ; 数値を文字列に変換
        mov     bx,OFFSET Result        ; 結果の文字列を指し示す
        call    PrintString             ; カウント値を表示
        mov     ah,4ch                  ; プログラム終了のDOSファンクション番号
        int     21h                     ; プログラム終了
;
; 標準入力から文字を取り込むサブルーチン
;
; 入力: なし
;
; 出力:
;         AL = 文字コード(文字ありの場合のみ)
;         ? flag = 0 (NZ) 文字あり
;                = 1 (Z)  ファイルの終わりに達したとき
;
; 破壊されるレジスタ:AH,BX,CX,DX
;
GetNestCharacter    PROC
        mov     ah,3fh                  ; ファイル読み込みのDOSファンクション番号
        mov     bx,0                    ; 標準入力のハンドルをセット
        mov     cx,1                    ; 1文字読み取る
        mov     dx,OFFSET TempChar      ; 文字をTempCharにストア
        int     21h                     ; 次の文字を読み取る
        jc      NoCharacterRead         ; DOSがエラーを通知してきたら,
                                        ; ファイルの終わりとして処理
        cmp     [TempChar],1ah          ; ^Z(ファイルの終わり)か?
        jne     NotControlZ             ; No
NoCharacterRead:
        sub     ax,ax                   ; 文字がないことを示す
        and     ax,ax                   ; Zフラグをセット. 文字を読み取れたら
                                        ; (NZ), ファイルの終わりに到達したら(Z)
                                        ; をセット. DOSファンクション#3fhは,
                                        ; 読み取った文字数をAXにセット
        mov     al,[TempChar]           ; 読み込んだ文字を復帰
        ret                             ; リターン
GetNextCharacter        ENDP
```

```
;
; 与えられた文字がホワイトスペースか否かをチェックするサブルーチン
;
; 入力:
;        AL = チェックする文字コード
;
; 出力:
;        z flag = 0 (NZ) ホワイトスペースでない
;               = 1 (z)  ホワイトスペース
;
; 破壊されるレジスタ:なし
;
IsCharacterWhitespace   PROC
        cmp     al,''                        ; スペースか?
        jz      EndIsCharacterWhitespace     ; Yes...ホワイトスペース
        cmp     al,09h                       ; タブか?
        jz      EndIsCharacterWhitespace     ; Yes...ホワイトスペース
        cmp     al,d9h                       ; 復帰コードか?
        jz      EndIsCharacterWhitespace     ; Yes...ホワイトスペース
        cmp     al,0ah                       ; 改行コードか?
                                             ; Yes...ホワイトスペース. Zを返す.
                                             ; ホワイトスペースでなければcmpで
                                             ; セットされたNZを返す
EndIsCharacterWhitespace:
        ret                                  ; リターン
IsCharacterWhitespace   ENDP
;
; バイナリを文字列に変換するサブルーチン
;
; 入力:
;       AX    = 変換するバイナリ
;       DX:BX = テキストをストアする文字列の終端を示すポインタ
;       CX    = 変換する桁数
;
; 出力: なし
;
; 破壊されるレジスタ: AX,BX,CX,DX,SI
;
ConvertNumberToString   PROC
        mov     si,10                        ; ConvertLoopを10で分割するために使用
        sub     dx,dx                        ; AXをDX:AXのダブルワードに変換
        div     si                           ; 数値を10で割る. 余り(1桁10進数)は
                                             ; DXにストア. 商はAXにストア.
        add     dl,0                         ; 余りを文字列に変換
        mov     [bx],dl                      ; 数字を文字列に挿入する
        dec     bx                           ; 次の最上位桁の位置を指し示す
        loop    ConvertLoop                  ; 次の桁があれば処理する
        ret
ConvertNumberToString   ENDP
```

```
;
; 画面に文字列を表示するサブルーチン
;
; 入力:
;       DS:BX = 表示文字列のポインタ
;
; 出力: なし
;
; 破壊されるレジスタ: なし
;
PrintString     PROC
        push    ax
        push    dx                      ; このサブルーチン内で使うレジスタを退避
        mov     ah,9                    ; 文字列表示のDOSファンクション番号
        mov     dx,bx                   ; DS:DXで表示文字列をポイント
        int     21h                     ; DOSファンクションコール
        pop     dx                      ; レジスタ復帰
        pop     ax
        ret
PrintString     ENDP
END     ProgramStart
```

WCOUNT.EXEを実行するためには，DOSのプロンプトで目的の(単語をカウントしたい) ファイルを指定します。つまり標準入力をそのファイルにリダイレクトします。たとえばWCOUNT.ASMというファイルの単語数をカウントしたければ，DOSプロンプトで次のようにタイプします。数秒後に結果が表示されます。

```
A:¥TA>wcount < wcount.asm
Word count: 874
```

このWCOUNT.ASMに関してはいくつかのポイントがあります。1つは，WCOUNT.ASMでは，文字の読み取り，文字がホワイトスペースか否かのチェック，カウントした数値の文字列への変換，文字列の表示などの細かい処理のためにサブルーチンを使っていることです。このため，WCOUNT.ASMのメインプログラムはコンパクトでわかりやすくなっています。

サブルーチンを使うことのもう1つの利点は，プログラムの動作を容易に変更できることです。たとえばホワイトスペースに等号（=）を加える場合は，メインのプログラムはそのままで*IsWhitespace*サブルーチンだけを変更すればよいのです。

*GetNextCharacter*も*IsWhitespace*も，ゼロフラグにステータス情報をセットしてリターンしていることに注意してください。*GetNextCharacter*は，文字をALにストアして返しています。ゼロフラグはYes/Noのステータスを返すのに適し，AL(またはAX)は値を返すのに適しています。

アセンブラで文字列を表示するには，いかに多くのステップが必要になるか注目してください。単語のカウント結果を整数で表示するためには，カウント値を10で割ってその余りに文字'0'を加える操作を繰り返して，カウント値を文字列に変換しなければなりません。

こうしないとDOS関数で文字列を表示させることができません。この処理は，Cでは次のような簡単な1文で済んでしまいます。

```
printf("Word count: %d¥n",Count);
```

しかし，*ConvertNumberToString*のようなサブルーチンをいったん作ってしまえば，必要に応じて他のプログラムから何度でも利用することができます。したがって役に立つアセンブラサブルーチンのライブラリを作っておくと，プログラム開発に要する時間と手間を省くことができます。

第5章

Turbo Assemblerプログラミング入門

第3章と第4章でアセンブリ言語の基礎的なことがらを学習しましたが，学ぶべきことはまだたくさんあります。この章はTurbo Assemblerの入門編です。アセンブリ言語プログラミングのさらに高度で有益な使用法を説明します。

この章で説明するTurbo Assemblerの機能は次のとおりです。

■文字列に名前を付ける疑似命令EQUと＝
■強力なストリング命令
■複数ソースファイルの個別アセンブル機能，TLINKを使ったオブジェクトファイルのリンクとプログラムの作成
■アセンブラプログラムへのソースコードファイルのインクルード機能
■高度なソースファイルのリスト機能

疑似命令を使えば，特定の条件に基づいてアセンブルするようにアセンブラプログラム内に指定できます。特定の条件によるアセンブルが役立つ理由とそのために必要な疑似命令についてもこの章で説明します。最後に，アセンブラプログラマの誰もが陥りやすい共通の落とし穴についても紹介します。

たとえ書いてあることのすべてが理解できなくても，この章にはひととおり目を通しておいてください。この章の内容はすぐに必要とはならないかもしれませんが，一度目を通しておけば，必要なときに読みたい箇所をすぐに見つけられるでしょう。

イコール定義

EQU疑似命令と=疑似命令を使えば，数値や文字列にラベルを割り当てることができます。この機能はアセンブルプログラムを明瞭にし，かつ保守しやすくするために非常に役に立ちます。

EQU疑似命令

変数，サブルーチン，命令などに名前を付けるためにラベルを使う理由は明らかです。これらに名前がなければ，プログラム要素を命令オペランドとして参照できません。これほど明白ではないにしろ，数値や文字列に割り当てるラベルも重要なものです。

EQU疑似命令を使えば，数値や文字列にラベルを割り当てることができます。また*EQU*ラベルを使うと，そのラベルの**リテラル**に変換されます。次にプログラム例を示します。

```
      .
      .
END_OF_DATA         EQU  '!'
STORAGE_BUFFER_SIZE EQU  1000
      .DATA
StorageBuffer  DB  STORAGE_BUFFER_SIZE DUP (?)
      .
      .
      .
.CODE
      mov  ax,@data
      mov  ds,ax
      sub  di,di                    ; バッファポインタを0にセット
StorageLoop:
      mov  ah,1
      int  21h                      ; 次のキーを読み込み
      mov  [StorageBuffer+di],al    ; キーコードをセーブ
      cmp  al,END_OF_DATA           ; end-of-dataキーか？
      je   DataAcquired             ; Yes, データを処理
      inc  di                       ; キー押下をカウント
      cmp  di,STORAGE_BUFFER_SIZE   ; バッファはオーバーフローしたか？
      jb   StorageLoop              ; No, 次のキーへ
; バッファがオーバーフローした場合の処理...
      .
      .
; データが得られた場合の処理
DataAcquired:
      .
      .
      .
```

この例では，**EQU**は*STORAGE_BUFFER_SIZE*と*END_OF_DATA*の2つのラベルを定義するために使われています。*END_OF_DATA*のラベルは文字"!"にイコール定義され，キー押下のたびにデータの終わりに達したかどうかチェックされます。この例はイコール定義を使うメリットをよく表わしています。つまり，定数を使うよりラベルを使った方がはるかに融通性が良くなります。たとえば

```
cmp  al,END_OF_DATA
```

と書く方が

```
cmp  al,'!'
```

と書くよりもわかりやすくなります。

*STORAGE_BUFFER_SIZE*はイコール定義のもう1つのメリットを表わしています。1000という数値に割り当てられた*STORAGE_BUFFER_SIZE*は，1000バイトのメモリバッファ作成とバッファオーバフローのチェックのために使われています。もちろんこれらの目的のために1000という数値を使ってもよいのですが，*STORAGE_BUFFER_SIZE*というラベルを使う方が融通性が良くなります。

ここでメモリバッファのサイズを変更するとしましょう。このためには**EQU**疑似命令のオペランドを変更するだけでよいのです。たったこれだけの操作で，プログラム中にあるメモリバッファのサイズをすべて変更したことになります。この例ではこれらの2つの数値を直接変更するのはそれほど大変なことではありません。しかしモジュール内に数十，数百の変更箇所がある場合は，イコール定義を使えば簡単に（しかもエラーなしで）値を変更することができるのです。

イコール定義のオペランドには，通常のラベルの他，イコール定義ラベルも指定することができます。次に例を示します。

```
.
.
.
TABLE_OFFSET   EQU  1000h
INDEX_START    EQU  (TABLE_OFFSET+2)
DICT_START     EQU  (TABLE_OFFSET+100h)
.
.
.
mov  ax,WORD PTR (bx+INDEX_START)    ; 最初のインデックス項目入手
.
.
.
lea  si,[bx,DICT_START]              ; 最初の辞書項目をポイント
.
.
.
```

上の例は次と同じです。

```
.
.
.
mov   ax,WORD PTR [bx+1000h+2]
lea   si,[bx+1000h+100h]
.
.
.
```

イコール定義のラベルは，コンピュータの多数の割り込み，ポート，メモリアドレスなどにわかりやすい名前を付ける便利なものです。次にEQUの使用例を示します。

```
      .
      .
      .
DOS_INT         EQU  21h          ; DOSファンクションの割込み
CGA_STATUS      EQU  3dah         ; CGAステータスポート
VSYNC_MASK      EQU  00001000b    ; 画面をスノーノイズなしで更新できる
                                 ;タイミングを知らせるCGAステータスポート
                                 ;のビットを隔離
BIOS_SEGMENT    EQU  40h          ; BIOSがデータをストアするセグメント
EQUIPMENT_FLAG EQU  10h          ; 装置フラグ変数のBIOSセグメント内のオフセット
      .
      .
      .
      mov  ah,2
      mov  dl,'Z'
      int  DOS_INT                ; 'z'をプリント
      .
      .
      .
; 画面をスノーノイズなしで更新できるまで待つ
      mov  dx,CGA_STATUS
WaitForVerticalSync:
      in   al,dx                  ; CGAステータス読み込み
      and  al,VSYNC_MASK          ; 垂直同期
      jz   WaitForVerticalSync    ; No,もう少し待つ
      .
      .
      .
      mov  ax,BIOS_SEGMENT
      mov  ds,ax                  ; DSにBIOSデータセグメントをセット
      mov  bx,EQUIPMENT_FLAG      ; セグメントフラグをセット
      and  BYTE PTR [bx],NOT 30h
      or   BYTE PTR [bx],20h      ; 80桁カラーモードを選択するように
                                  ; 装置フラグをセット
      .
      .
      .
```

他のイコール定義に基づいたイコール定義を使えば，容易にプログラムを変更すること

ができます。先の例ですべてのデータをBXが示すアドレスにもう10バイトだけ近いテーブルに転送したい場合は，*TABLE_OFFSET*のイコール定義を次のように変更すればよいのです。

```
TABLE_OFFSET    EQU  (1000h-10)
```

*INDEX_START*と*DICT_START*の値は*TABLE_OFFSET*に基づいているため，これらは両方とも*TABLE_OFFSET*のイコール定義にしたがって変更されます。

ところで**EQU**疑似命令のオペランドのカッコは本来必要ありません。しかしカッコを付けてオペランドを区切った方がプログラムが見やすくなります。

EQUは数値だけでなく文字列にも使うことができます。たとえば次のプログラムでは，プリント文字列に対してイコール定義を使っています。

```
      .
      .
      .
EQUATED_STRING EQU 'This text started life in an EQU directive$'
      .
      .
      .
TextMessage    DB   EQUATED_STRING
      .
      .
      .
      mov   dx,OFFSET TextMessage
      mov   ah,9
      int   21h                          ; TextMessageをプリント
      .
      .
      .
```

次のように文字列のイコール定義ラベルをオペランドとして使うことができます。

```
      .
      .
      .
REGISTER_BX    EQU   BX
      .
      .
      .
      mov   ax,REGISTER_BX
      .
      .
      .
```

この例は次の命令と同様にアセンブルされます。

```
      mov   ax,bx
```

レジスタ名に対してイコール定義を利用してもあまりメリットはありません。しかし，スタックに入れるパラメータやスタック上のダイナミックメモリ領域に名前を付けるためにイコール定義ラベルやARGを使うことは意味があります。次に例を示します。

```
;
; 3つのintパラメータを加え，intの結果を返す，Cからコールされるnearモデルサブ
; ルーチン呼び出し形式：
;
; int AddThree(int I, int J, int K)
;
Temp EQU [bp-2]
I    EQU [bp+4]
J    EQU [bp+6]
K    EQU [bp+8]
;
_AddThree PROC
     push bp                  ; 呼び出し側のBP退避
     mov  bp,sp               ; スタックポインタをロード
     sub  sp,2                ; Temp用スペースを確保
     mov  ax,I                ; Iをロード
     add  ax,J                ; I+J
     mov  Temp,ax             ; I+Jをセーブ
     mov  ax,K                ; Kをロード
     add  ax,Temp             ; I+J+K
     mov  sp,bp               ; Temp用スペースを開放
     pop  bp                  ; 呼び出し側のBP復帰
     ret
_AddThree ENDP
```

基本的にはEQU定義を使えば，文字列に名前を付けたり文字列をオペランドとして使うことができます。思いがけないことかもしれませんが，オペランドフィールドだけでなく命令や疑似命令のフィールドにもイコール定義ラベルを使うことができます。

不等号カッコ（<>）を使えば，EQUのオペランドに式ではなく文字列を指定することができます。次の例では，*INDEX_START*というラベルに文字列"TABLE_OFFSET+2"を割り当てます。

```
TABLE_OFFSET   EQU 1
INDEX_START    EQU <TABLE_OFFSET+2>
```

次の例は*INDEX_START*というラベルに数値3（1+2の結果）を割り当てます。

```
TABLE_OFFSET   EQU 1
INDEX_START    EQU TABLE_OFFSET+2
```

EQU定義の文字列は常に<>で囲む習慣をつけておけば，間違って式として評価されるのを防ぐことができます。

EQUを使って数値や文字列に割り当てたラベルを同一モジュール内で再定義することは

許されません。したがって次の例はエラーとなります。

```
     .
     .
     .
X    EQU  1
     .
     .
     .
X    EQU  101
     .
     .
     .
```

イコール定義のラベルを再定義する場合は，＝疑似命令を使う必要があります(ラベルを再定義すると便利な場合もあります)。＝疑似命令については後で簡単に説明します。

$定義済みシンボル

Turbo Assemblerには@**data**などの定義済みシンボルがあります。他に便利な定義済みシンボルとして$があります。$には常にロケーションカウンタの現在値がセットされています。つまり$は常に（Turbo Assemblerが現在アセンブルしている）セグメント内のカレントオフセットの値を保持しています。$は**OFFSET** *MenuVar*と同様にオフセット定数であるため，式中または定数を伴う場所ならどこにでも使うことができます。

$はデータ長やコード長を計算するのに非常に便利です。たとえば文字列のバイト長に*STRING_LENGTH*というラベルを割り当てたい場合，$を使わないと次のようになります。

```
       .
       .
       .
StringStart    LABEL    BYTE
     db   0dh,0ah,'Hello, world',0dh,0ah
StringEnd   LABEL    BYTE
STRING_LENGTH   EQU [StringEnd-StringStart]
       .
       .
       .
```

$を使うと次のようになります。

```
      .
      .
      .
StringStart   LABEL  BYTE
      db  0dh,0ah,'Hello, world',0dh,0ah
STRING_LENGTH  EQU [$-StringStart]
      .
      .
      .
```

次の例はワード配列のワード数を$を使って計算します。

```
      .
      .
      .
WordArray DW  90h, 25h, 0, 16h, 23h
WORD_ARRAY_LENGTH  EQU  (($-WordArray)/2)
      .
      .
      .
```

もちろん要素の1つ1つを数えてもよいのですが，長い配列や文字列を数えることは退屈な作業です。

また他にも次のような3種類の定義済みシンボルがあります。

??Date：アセンブルの日付の文字列（*01/02/87*など）を表わす。
??Time：アセンブルの時刻の文字列（*13:45:06*など）を表わす。
??Filename：アセンブルされているファイル名が*8*文字形式（*"TEST.ASM"*など）を表わす。

＝疑似命令

＝疑似命令は，一点だけを除いて**EQU**疑似命令とよく似ています。**EQU**疑似命令で定義したラベルは再定義できません(再定義するとエラーとなります)。しかし＝疑似命令で定義したラベルは自由に再定義することができます。これは途中で変更する必要のあるラベルや同一ソースモジュール内で再利用するラベルに対しては非常に便利です。

次のプログラムは，＝疑似命令を使って100個の10の倍数のルックアップテーブルを作成します。

```
      .
      .
      .
      .Data
MultiplesOf10   LABEL   WORD
TEMP =    0
      REPT 100
      DW    TEMP
TEMP =    TEMP+10
      ENDM
      .
      .
      .
      shl   bx,1                    ; BXは10倍される数
                                    ; ワードサイズテーブル参照のために
                                    ; 左へ1ビットシフトして2倍する
      mov   ax,[MultiplesOf10+bx]   ; 数値*10
      .
      .
      .
```

＝疑似命令のオペランドは最終的に数値として評価されなければなりません。したがって＝疑似命令で文字列にラベルを割り当てることはできません。これも**EQU**疑似命令と異なる点です。

ストリング命令

この節では，8086のもっとも非凡で強力な命令であるストリング命令を説明します。ストリング命令は1命令でメモリアクセスとポインタレジスタのインクリメント(またはデクリメント)を同時に行なうという，他の8086命令にはない特長をもっています。1つのストリング命令で最大約130,000回メモリにアクセスすることができます(！)。

その名が示すとおり，ストリング命令は特に文字列処理に威力を発揮します。また配列，データバッファ，バイトまたはワードサイズの文字列などの処理にも効果的です。プログラムを作るときはできるだけストリング命令を使うことをお勧めします。ストリング命令を使えば，**MOV, INC, LOOP**などの通常の8086命令を組み合わせて使うよりステップ数が減って処理速度は上がります。

ストリング命令は，その機能からデータ転送命令（**LODS, STOS, MOVS**）とデータスキャンおよび比較命令（**SCAS, CMPS**）の2種類に分類することができます。

データ転送ストリング命令

データ転送ストリング命令は**MOV**命令と似ていますが，**MOV**命令より機能的でしかも実行速度が速くなっています。はじめに**LODS**を説明します。すべてのストリング命令に対するポインタレジスタの方向は，ディレクション（方向）フラグによってコントロールされます。

LODS命令

この命令はメモリからアキュムレータへ1バイト（**LODSB**）または1ワード（**LODSW**）を転送します。**LODSB**はDS：SIによって指されたバイトをALレジスタに転送し，ディレクションフラグにしたがってSIレジスタをインクリメント（またはデクリメント）します。SIレジスタはディレクションフラグが0のときはインクリメントされ，1のときはデクリメントされます(なおディレクションフラグは**CLD**によって0にクリアされ，**STD**によって1にセットされます)。このようなディレクションフラグによるポインタレジスタの方向コントロールは，**LODSB**命令に限らずすべてのストリング命令で同じように行なわれます。

次に**LODSB**命令の使用例を示します。

```
    .
    .
    .
    cld
    mov  si,0
    lodsb
    .
    .
    .
```

この例は，ALレジスタにデータセグメントのオフセット0から1バイトをロードし，SIレジスタをインクリメントして1にします。次のプログラムも同じ処理を行ないます。

```
    .
    .
    .
    mov  si,0
    mov  al,[si]
    inc  si
    .
    .
    .
```

ただし

```
    lodsb
```

の方が

```
    mov  al,[si]
    inc  si
```

よりも処理速度が速く（しかも2バイト短かく）なります。

LODSWも**LODSB**と同様です。ただし，LODSWはDS：SIで指されたワードをAXレジスタにロードし，SIレジスタを＋2または－2します。次のプログラムはオフセット10のワードをAXレジスタにロードし，SIレジスタの内容を-2して8にします。

```
    .
    .
    .
    std
    mov  si,10
    lodsw
    .
    .
    .
```

STOS命令

STOSの働きは**LODS**の逆です。つまりアキュムレータの内容をES：DIで指されるメモリアドレスに転送し，DIの内容をインクリメントまたはデクリメントします。**STOSB**はALレジスタのバイトをES：DIで指されたメモリアドレスにストアし，ディレクションフラグにしたがってDIの内容をインクリメントまたはデクリメントします。次のプログラムは55hをESレジスタで指定されるセグメント内のオフセット0FFFFhにストアし，DIレジスタの内容をデクリメントして0FFFEhにします。

```
.
.
.
std
mov  di,0ffffh
mov  al,55h
stosb
.
.
.
```

STOSWはAXレジスタのワードをES：DIで指定されたメモリアドレスにストアし，ディレクションフラグにしたがってDIレジスタの内容を 2または-2します。次のプログラムはAXレジスタの102hをESレジスタで指定されるセグメント内のオフセット0FFEhにストアし，DIレジスタの内容をインクリメントして1000hにします。

```
.
.
.
cld
mov  di,0ffeh
mov  ax,102h
stosw
.
.
.
```

LODS命令も**STOS**命令もバッファの転送を行なうときに便利です。たとえば次のサブルーチンは，DS：SIの0で指される文字列をES：DIで指されるメモリアドレスへ転送します。

```
;
; 0で終わるの文字列を他の領域に転送するサブルーチン
;
; 入力:
;       DS:SI = ソースアドレス
;       ES:DI = デスティネーションアドレス
;
; 出力:なし
;
; 破壊されるレジスタ : AL, SI, DI
;
CopyString    PROC
    cld                         ; ディレクションフラグをクリア(SIとDIを
                              ; ストリング命令でインクリメントするように設定する)
CopyStringLoop:
     lodsb                      ; ソースから文字列をロード
     stosb                      ; デスティネーションアドレスへストア
     cmp  al,0                  ; 0(文字列の終わり)か?
     jnz  CopyStringLoop        ; No, 次のデータへ
     ret                        ; Yes, リターン
CopyString   ENDP
```

同様にLODSとSTOSを使って，終端が0でないブロックをワードサイズで転送することができます。次に例を示します。

```
     .
     .
     .
     mov  cx,ARRAY_LENGTH_IN_WORDS   ; ワード数
     mov  si,OFFSET SourceArray      ; ソースオフセット
     mov  ax,SEG SourceArray         ; ソースセグメント
     mov  ds,ax
     mov  di,OFFSET DestArray        ; デスティネーションオフセット
     mov  ax,SEG DestArray           ; デスティネーションセグメント
     mov  es,ax
     cld
CopyLoop:
     lodsw                           ; ソースから文字列をロード
     stosw                           ; 文字列をストア
     loop  CopyLoop                  ; 次のデータへ
     .
     .
     .
```

メモリ間でバイトまたはワードサイズでデータ転送を行なう場合は，次の**MOVS**命令を使うのがよいでしょう。

MOVS命令

MOVS命令は**LODS**と**STOS**命令を1つにしたような命令です。つまりDS：SIで示されるバイトまたはワードをES：DIで示されるメモリアドレスへレジスタを経由することなく直

接転送します。したがってAXレジスタの内容は変化しません。**MOVSB**は1バイト命令で，**LODS**と**STOS**を組み合わせた場合より高速です。**MOVS**を使えば先のプログラムはより高速になります。次のプログラムは，**MOVS**を使ってさきほどのプログラム例を書き直したものです。

```
      .
      .
      .
      mov   cx,ARRAY_LENGTH_IN_WORDS    ; ワード数
      mov   si,OFFSET SourceArray       ; ソースオフセット
      mov   ax,SEG SourceArray          ; ソースセグメント
      mov   ds,ax
      mov   di,OFFSET DestArray         ; デスティネーションオフセット
      mov   ax,SEG DestArray            ; デスティネーションセグメント
      mov   es,ax
      cld
  CopyLoop:
      movsw                             ; 1ワード転送
      loop  CopyLoop                    ; 次のデータへ
      .
      .
      .
```

ストリング命令の繰り返し実行

上の例はたいへん効率のよいプログラムに見えます。この例で**LOOP**命令を取り去って1つの命令で配列全体を転送できたら便利です。そのような目的のために，8086のストリング命令にはオプションとして**REP**プレフィクスが用意されています。

REPそれ自身は命令ではなく，**命令プレフィクス**です。命令プレフィクスは後続命令の動作をコントロールします。**REP**の機能はCXレジスタの内容が0になるまで後続ストリング命令の実行を繰り返すことです(はじめからCXが0のときは命令は0回実行されます。つまり何も行ないません)。

REPプレフィクスを使えば，上のプログラムの

```
  CopyLoop:
      movsw
      loop CopyLoop
```

の3行は次の1行に置き換えることができます。

```
  rep   movsw
```

この例では1命令で65,535(0FFFFh)ワードのデータをDS：SIで指定されるアドレスからES：DIで指定されるアドレスへ転送します。

ここではストリング命令を65,535回も繰り返して実行する(つまりメモリにアクセスする)わけですから，1つのストリング命令の実行に比べたら余計に時間がかかります。しかし1回の転送に要する時間は1つのストリング命令よりは短かくなります。したがってメモリのアクセスや転送の際にREPを使えば，非常に高速に処理することができます。

REPプレフィクスは**MOVS**命令だけでなく**LODS**命令や**STOS**命令にも使うことができます(**SCAS**命令や**CMPS**命令に対しても使えます。これらの命令については次に説明します)。メモリブロックをクリアしたりパディング（充塡）する際は，**STOS**命令を繰り返して実行するのが効果的です。次に例を示します。

```
.
.
.
cld
mov  ax,SEG WordArray
mov  es,ax
mov  dl,OFFSET WordArray
sub  ax,ax
mov  cx,WORD_ARRAY_LENGTH
rep  stosw
.
.
.
```

この例では*WordArray*を*0*でパディングします。**LODS**命令の繰り返し処理の例として，これほど適当な例は他にありません。

REPが使える命令はストリング命令だけです。次のような使いかたはできません。

```
rep  mov  al,[bx]
```

この場合は**REP**プレフィクスは無視され，次の命令として扱われます。

```
mov  al,[bx]
```

文字列ポインタのオーバラン

ストリング命令を実行すると，メモリがアクセス**された後に**SIレジスタ，DIレジスタ，またはその両方がインクリメントまたはデクリメントされます。したがって命令実行後のポインタレジスタは，たった今アクセスされたアドレスではなく次にアクセスされるアドレスを指しています。このことは上記のプログラム例でも示すとおり，効果的にループ処理を行なえるため便利です。しかし次に説明するデータスキャンストリング命令では混乱をきたすこともあります。

データスキャンストリング命令

これまではデータ転送に関するストリング命令を学習しました。ここではデータスキャン用のストリング命令（**SCAS, CMPS**）を説明します。これらの命令はメモリブロックのスキャンや比較のために使います。

SCAS命令

SCASはメモリをスキャンして，特定のバイトまたはワードデータの一致または不一致を見つけるための命令です。他のストリング命令と同様，**SCAS**にも**SCASB**と**SCASW**の2種類があります。

SCASBはALレジスタの内容をES：DIで指定されるアドレスの1バイトと比較し，その結果に基づいてフラグをセットします(**CMP**命令と同様です)。**STOSB**と同様，**SCASB**実行の際もDIレジスタの内容がインクリメントまたはデクリメントされます。次のプログラムは文字列*TextString*内の特定の小文字を検索します。

```
      .
      .DATA
TextString          DB   'Test text',0
TEXT_STRING_LENGTH  EQU  ($-TextString)
      .
      .
      .
      .CODE
      .
      .
      .
      mov  ax,@data
      mov  es,ax
      mov  di,OFFSET TextString        ; ES:DIがTextStringの先頭を指す
      mov  al,'t'                      ; スキャンする文字
      mov  cx,TEXT_STRING_LENGTH       ; スキャン文字列の長さ
      cld                              ; スキャンしてDIをインクリメント
Scan_For_t_Loop:
      scasb                            ; ES:DIはALと一致したか?
      je   Found_t                     ; Yes, "t"を発見
      loop Scan_For_t_Loop             ; No,次の文字をスキャン
; "t" が見つからない場合
      .
      .
      .
; "t" が見つかった場合
Found_t:
      dec  di                          ; "t"のオフセットへ戻す
```

この例では文字"*t*"が見つかるとDIレジスタの内容をデクリメントします。これは先に説明した文字列ポインタのオーバランです。このプログラムを実行して最後の**SCASB**命令を実行すると，DIレジスタは文字"*t*"の次のバイトを指し，そのためデータの比較後にDIレジスタをデクリメントする必要があります（ストリング命令実行後には必らずポインタがインクリメントまたはデクリメントされることはすでに述べました)。オーバランを補正して文字tを指したければDIレジスタを調整しなければなりません。

上記プログラムの**SCASB**の処理をストリング命令を使わずに行なうと，次のようになります。

```
      .
      .
      .
Scan_For_t_Loop:
      cmp  es:[di],al              ; ES:DIがALと一致するか？
      je   Found_t                 ; Yes，"t"を発見
      inc  di
      loop Scan_For_t_Loop         ; No，次の文字をスキャン
      .
      .
      .
```

この例は先の**SCASB**の例とは少し異なっています。**SCASB**命令はDIをすぐにインクリメントします。しかしこの例では**JE**命令の後にDIをインクリメントする命令を置いて**CMP**によるフラグの変化を防いでいます。

これはストリング命令に関する一般的で重要な点です。ストリング命令を実行した後でSI, DI, CXなどのレジスタの内容が変化してもフラグには反映されません。また**LODS, STOS, MOVS**命令でもフラグは変化しません。しかし**SCAS**と**CMPS**命令を実行すると，その比較処理の結果によってフラグが変化します。

先の例でループを無くして1つの命令にすることができれば便利です。**REP**プレフィクスを使えばそれが可能となります。しかしデータの一致または不一致によってループを止めることが必要な場合もあります。**SCAS**（および**CMPS**）に対して使える**REP**プレフィクスは**REPE**と**REPNE**です。

REPE(または**REPZ**)はCXレジスタが0になるまで，または不一致が発生するまで**SCAS**（または**CMPS**）命令を繰り返し実行することを指定します。したがって**REPE**は「等しい限り繰り返す」ことを指定するプレフィクスと考えることができます。同様に，**REPNE**（または**REPNZ**）はCXレジスタが0になるまで，または一致が発生するまで**SCAS**（または**CMPS**）命令を繰り返し実行することを指定します。したがって**REPNE**は「等しくない限り繰り返す」ことを指定するプレフィクスと考えることができます。

SCASB命令を1回だけ使って*TextString*中の文字"*t*"を探すプログラムを次に示します。

```
    .
    .
    mov   ax,@data
    mov   es,ax
    mov   di,OFFSET TextString          ; ES:DIはTextStringの先頭を指す
    mov   al,'t'                        ; スキャンする文字
    mov   cx,TEXT_STRING_LENGTH         ; スキャンする文字の長さ
    cld                                 ; スキャンしてDIをインクリメント
    repne scasb                         ; 文字列全体をスキャンして文字"t"を探す
    je    Found_t                       ; Yes, "t"を発見
; "t"が見つからない場合
    .
    .
; "t"が見つかった場合
Found_t:
    dec   di                            ; "t"のオフセットに戻す
    .
    .
```

他のストリング命令と同様，**SCAS**命令でもディレクションフラグが0の場合はDIレジスタはインクリメントされ，1の場合はDIレジスタはデクリメントされます(なおディレクションフラグは**CLD**によって0にクリアされ，**STD**によって1にセットされます)。

SCASWはSCASBのワードサイズの命令で，AXレジスタの内容とES：DIで指定されるメモリアドレスの内容を比較し，そのたびにDIレジスタの内容を＋2または－2します。次の例は**REPE SCASW**命令を使ってワードサイズの整数配列内の0以外の最後の要素を見つけるプログラムです。

```
    .
    .
    .
    mov   ax,SEG ShortInArray
    mov   es,ax
    mov   di,OFFSET ShortInArray+((ARRAY_LEN_IN_WORDS-1)*2)
                                        ; ES:DIがShortIntArrayを指す
    mov   cx,ARRAY_LEN_IN_WORDS
    sub   ax,ax                         ; 0と一致しないものを探す
    std                                 ; 終りから逆順に探してDIをデクリメント
    repe scasw                          ; 0でないワードが見つかるまで探す
                                        ; または配列を終わる
    jne   FoundNonZero
; 配列全体が0だけだった場合
    .
    .
    .
; 0以外の要素を発見 - DIのオーバランを補正
FoundNonZero:
    inc   di
    inc   di
    .
    .
    .
```

CMPS命令

CMPS命令はバイトまたはワードの2つのデータを比較する命令です。1つの**CMPS**は2つのメモリアドレスの内容を比較し，SIとDIレジスタをインクリメントします。CMPSとMOVSはよく似ています。**CMPS**は2つのメモリアドレスを比較する命令で，**MOVS**は2つのメモリアドレス間でデータを転送する命令です。

CMPSB命令はDS：SIで指定されるメモリの内容とES：DIで指定されるメモリの内容を比較し，フラグをセットし，ディレクションフラグに基づいてSIレジスタとDIレジスタをインクリメントまたはデクリメントします。AXレジスタは影響を受けません。

他のストリング命令と同様，**CMPS**命令もバイトサイズとワードサイズがあります。またSIとDIレジスタとインクリメント（またはデクリメント）し，**REP**プレフィクスが付いていれば処理を繰り返します。次の例は2つのワード配列の先頭50個の要素が同一かどうかをチェックするプログラムです。**REP CMPSW**を使っています。

```
    .
    .
    .
    mov  si,OFFSET Array1
    mov  ax,SEG Array1
    mov  ds,ax
    mov  di,OFFSET Array2
    mov  ax,SEG Array2
    mov  es,ax
    mov  cx,50                   ; 先頭50要素まで比較
    cld
    repe cmpsw
    jne  ArraysAreDifferent
; 先頭の50要素が同一の場合
    .
    .
    .
; 1要素でも違っている場合
ArrayAreDifferent:
    dec  si
    dec  si                      ; 違っている要素にポインタを戻す
    dec  di
    dec  di
    .
    .
    .
```

ストリング命令のオペランド指定

これまではストリング命令のバイトサイズとワードサイズの命令，つまり**LODSB**と**LODSW**を見てきました。しかし**LODS**はまだ使っていません。オペランドによって処理がバイトサイズかワードサイズかが判明する限り，データサイズを明示しないストリング命令を使うことができます。

次に例を示します。この例は**MOVSB**と同様です。

```
      .
      .
      .
      .DATA
String1   LABEL   BYTE
    db    'abcdefghi'
STRING1_LENGTH EQU  ($-String1)
String2   DB    50 DUP (?)
      .
      .
      .
      .CODE
      mov   ax,@data
      mov   ds,ax
      mov   es,ax
      mov   si,OFFSET String1
      mov   di,OFFSET String2
      mov   cx,STRING1_LENGTH
      cld
      rep   movs es:[String2],[String1]
      .
      .
      .
```

この例では*String1*と*String2*を**MOVS**のオペランドとして指定しているため，**MOVS**命令のデータサイズはオペランドのデータサイズ（この場合はバイト）となります。

ストリング命令のオペランドの使用には気を付けなければならない点があります。ストリング命令のオペランドは命令に組み込まれており，実行時にSIレジスタまたはDIレジスタのどちらかまたは両方を使います。このためストリング命令のオペランドは実オペランドではありません。オペランドはデータサイズを設定するために使われ，ポインタをロードするためには使われません。たとえば

```
mov   al,[String1]
```

のような命令をアセンブルすると，*String1*のオフセットは**MOV**のマシンコード内に組み込まれます。しかし次の命令

```
lods [String1]
```

をアセンブルしても，得られるマシンコードは**LODSB**の1バイトだけです。この場合プログラマは自分の責任において，DS：SIが*String1*の先頭を指していることを確認しておかなければなりません。

ストリング命令に対するオペランドの役割は，セグメントに対する**ASSUME**疑似命令のそれとよく似ています。**ASSUME**疑似命令はセグメントレジスタに値を実際にセットするのではなく，プログラマがセグメントレジスタをどのようにセットしたかをTurbo Assemblerに通知します。Turbo Assemblerはその情報によってエラー チェックを行なうことができます。同様にストリング命令のオペランドもレジスタに値をセットするわけではなく，プログラマがSIやDIレジスタにセットしたものが何かをTurbo Assemblerに知らせるだけです。Turbo Assemblerはその情報によってオペランドサイズを決定しエラー チェックを行なうことができます。ストリング命令のオペランドの詳細についてはこの章の後半にある「ストリング命令のオペランドをあてにする」の節を参照してください。

章の後半にある「ストリング命令の落とし穴」の節では，ストリング命令の使用上の注意点を説明します。

マルチモジュールプログラム

遅かれ早かれプログラムのソースファイルを1つのファイルにしまっておくのは，何かと不都合であることに気づくでしょう。1ファイルのソースプログラムは，このマニュアルのサンプルプログラムのような短かいプログラムには適しています。しかし中規模のプログラムでは複数のファイルまたはモジュールに分割して，個別にアセンブルしてリンクする必要があります。マルチモジュールプログラム（複数のモジュールから構成されるプログラム）の基本的なメリットは，ソースプログラムを変更した場合はそのモジュールのみを再アセンブルすればよい点にあります。また1本の大きなファイルよりも複数の短かいファイルの方が管理がしやすくなります。

マルチモジュールプログラムの作り方は非常に簡単です。Turbo Assemblerにはマルチモジュールプログラムのサポートのための3種類の疑似命令，**PUBLIC, EXTRN, GLOBAL**が備っています。これらの各疑似命令の説明に入る前に，2つのモジュールから成るサンプルプログラムを見てみましょう。その方がこれらのマルチモジュール疑似命令がより理解しやすくなるでしょう。次のプログラムがメインプログラムMAIN.ASMです。

```
     DOSSEG
     .MODEL  SMALL
     .STACK  200H
     .DATA
String1      DB   'Hello, ',0
String1      DB   'World',0dh,0ah,'$',0
    GLOBAL   FinelString:BYTE
FinalString DB   50 DUP (?)
     .CODE
     EXTERN  ConcatenateStrings:PROC
ProgramStart:
     mov  ax,@data
     mov  ds,ax
     mov  ax,OFFSET String1
     mov  ax,OFFSET String2
     call ConcatenateStrings          ; 2つの文字列を連結
     mov  ah,9
     mov  dx,OFFSET FinalString
     int  21h                         ; 最終文字列をプリント
     END  ProgramStart
```

次のプログラムは別のモジュールSUB1.ASMです。

```
        DOSSEG
        .MODEL   SMALL
        .DATA
        GLOBAL   FinalString:BYTE
        .CODE
;
; FinalStringへ第1文字列，第2文字列の順番で転送するサブルーチン
;
; 入力：
;       DS:AX = 第1文字列へのポインタ
;       DS:BX = 第2文字列へのポインタ
;
; 出力： なし
;
; 破壊されるレジスタ ： AL,SI,DI,ES
;
        PUBLIC   ConcatenateStrings
ConcatenateStrings   PROC
        cld                                 ; 文字列カウントアップ指定
        mov   di,SEG FinalString
        mov   es,di
        mov   di,OFFSET FinalString         ; ES:DIで転送先を指定
        mov   si,ax                         ; String1Loop:転送のための第1文字列
        lodsb                               ; 第1文字列ロード
        and   al,al                         ; 0か？
        jz    DoString2                     ; Yes，第1文字列終了
        stosb                               ; 第1文字列セーブ
Dostring2:
        mov   si,bx                         ; String2Loop:転送のための第2文字列
        lodsb                               ; 第2文字列ロード
        stosb                               ; 第2文字列セーブ(0含む)
        and   al,al                         ; 0か？
        jnz   String2Loop                   ; No，次の文字へ
        ret                                 ; 終了
ConcatenateString    ENDP
        END
```

この2つのモジュールは，それぞれ次の2つのコマンドでアセンブルされます。

```
TASM main
```

および

```
TASM sub1
```

また次のコマンドによってリンクされ，MAIN.EXEというプログラムファイルが作られます。

```
tlink main+sub1
```

このプログラムを実行するには次のコマンドを入力します。

```
main
```

MAIN.EXEが実行され次のように表示されます。

```
Hello, world
```

マルチモジュールについて簡単に紹介しました。次に先ほどの3つの疑似命令を説明します。

PUBLIC疑似命令

PUBLIC疑似命令の機能は単純です。1つまたは複数のラベルを他のモジュールから参照することをTurbo Assemblerに通知します。**PUBLIC**を使えば，プロシージャ名，メモリ変数名，イコール定義ラベルなどのすべてのラベルは他のモジュールから参照可能になります。次にその使用例を示します。

```
     .
     .
     .
     .DATA
     PUBLIC   MemVar, Array1, ARRAY_LENGTH
ARRAY_LENGTH  EQU  100
MemVar        DW   10
Array1        DB   ARRAY_LENGTH DUP (?)
     .
     .
     .
     .CODE
     PUBLIC    NearProc, FarProc
NearProc  PROC NEAR
     .
     .
     .
NearProc  ENDP
     .
     .
     .
FarProc  LABEL     PROC
     .
     .
     .
     END
```

イコール定義ラベル名，ワード変数名，配列名，nearプロシージャ名，farプロシージャ名などは，このプログラムとリンクされる他のモジュールから参照することができます。

パブリックラベルとして使えないラベルもあります。それは1バイトまたは2バイトの定

数以外に割り当てられたイコール定義ラベルです。たとえば次の2つのラベルがそれに該当します。

```
LONG_VALUE    EQU   100000h
TEXT_SYMBOL   EQU   <TextString>
```

Turbo Assemblerはラベルの大文字小文字の区別はつけません。パブリックラベルはアセンブルの際にすべて大文字に変換されます。パブリックラベルの大文字小文字の区別をつけたい場合は，そのパブリックラベルを参照するすべてのモジュールで/**ML**または/**MX**のコマンドラインスイッチを使ってください。

/**ML**または/**MX**のスイッチを付けない場合は，Turbo Assemblerはたとえば次の2つのラベルは区別せず，同一ラベルと見なします。

```
PUBLIC    Symbol1, SYMBOL1
```

/**MX**コマンドラインスイッチを使えば，パブリックシンボルや外部シンボルに対して大文字小文字の区別をつけます。この場合は**PUBLIC**や**EXTRN**疑似命令でシンボル名を記述する際に大文字小文字の区別に注意しましょう。**EXTRN**や**PUBLIC**疑似命令内で指定されたシンボルは，他のモジュールからでも使用できます。これはラベルの形式，定義場所，参照場所には関係しません。次の例では*Abc*はパブリックラベルですが，*abC*はそうではありません。

```
      PUBLIC Abc
abC   Dw
```

EXTRN疑似命令

前の節では，**PUBLIC**疑似命令を使えば*MemVar, Array1, Array_Length, NearProc, FarProc*などのラベルを他のモジュールから参照できることを説明しました。では他のモジュールはそれらのラベルをどのように参照するのでしょうか？

EXTRN疑似命令を使えば，任意のモジュール内のラベルが他のモジュールから参照可能となり，あたかもモジュール内でそのラベルが定義されたかのように使うことができます。次のプログラムは，**EXTRN**疑似命令を使って前の節で定義したパブリックラベルを参照します。

```
.
.
.
.DATA
EXTRN    MemVar:WORD,Array1:BYTE,ARRAY_LENGTH:ABS
.
.
.
.CODE
EXTRN    NearProc:NEAR,FarProc:FAR
.
.
.
mov   ax,[MemVar]
mov   bx,OFFSET Array1
mov   cx,ARRAY_LENGTH
.
.
.
call NearProc
.
.
.
call FarProc
.
.
.
```

5種類のラベルはすべて通常と同じように使われています。この例では，**EXTRN**疑似命令がある点が単一モジュールのアセンブラプログラムと異なっています。

EXTRN疑似命令を使って宣言したラベルは，その直後にコロンで区切って型を指定します。Turbo Assemblerはラベルの型を知る手立てがないため，プログラマがその型宣言をしてやらなくてはなりません。外部ラベルとして使える型と**LABEL**疑似命令で使える型は同じです。使用できる型は次のとおりです。

ABS	絶対値
BYTE	バイトサイズのデータ変数
DATAPTR	現在のメモリモデルに従ったデータポインタ（nearまたはfar）
DWORD	ダブルワードサイズ（4バイト）のデータ変数
FAR	farコードラベル（CS：IPの値に基づいてジャンプする）
FWORD	6バイトのデータ変数
NEAR	nearコードラベル（IPの値に基づいてジャンプする）
PROC	プロシージャコードラベル（**.MODEL**に従ってnearまたはfarとなる）

QWORD	4ワード（8バイト）のデータ変数
TBYTE	10バイトのデータ変数
UNKNOWN	未定型
WORD	2バイトのデータ変数

これらのデータ型のうち**ABS**は唯一見慣れないものです。この型は元のモジュール内で**EQU**や**=**で定義されるラベルを宣言するために使われます。つまり定数に対するラベルであり，コードアドレスとデータアドレスには関係していません。

外部ラベルのデータ型を正しく宣言することは重要なことです。なぜならTurbo Assemblerは，指定されたデータ型に基づいてコードを生成するからです。データ型を誤って指定してもTurbo Assemblerはそれを認識できません。次に誤ったソースコードの入力例を示します。

```
.
.
.
.CODE
EXTRN    FarProc:NEAR
.
.
.
call FarProc
.
.
.
```

他のモジュールのソースは次のようになっているとします。

```
      .
      .
      .
      PUBLIC  FarProc
FarProc   PROC FAR
      .
      .
      .
      ret
FarProc   ENDP
      .
      .
      .
```

この例では，**EXTRN**で指定したデータ型に従って，Turbo Assemblerはラベル*FarProc*に対してnearコールを生成します。しかし*FarProc*は実際はfarプロシージャにもかかわらず，最後が**RET**命令で終わっているためうまく動きません。

前の節で説明したように，通常Turbo Assemblerはラベルの大文字小文字を区別せず，パブリックラベルはすべて大文字に変換します。したがって外部ラベルには通常大文字を使います。外部ラベルの大文字小文字の区別をつけたい場合は，コマンドラインで/**ML**または/**MX**スイッチを指定してください。

GLOBAL疑似命令

モジュール間でラベルを共有するためには**PUBLIC**と**EXTRN**の2つの疑似命令が必要です。この理由は他のアセンブラとの互換性を保つことです。Turbo Assemblerには，**PUBLIC**と**EXTRN**の両方を兼ねる**GLOBAL**という疑似命令があります。

ラベルをグローバル宣言して，かつモジュール内で（**DB, DW, PROC, LABEL**などによって）定義した場合は，そのラベルは（**GLOBAL**ではなく）**PUBLIC**を使ったときと同じように他のモジュールから参照されます。一方ラベルのグローバル宣言だけを行ない，モジュール内で定義していない場合は，そのラベルは**EXTRN**を使ったときと同じように外部ラベルとして扱われます。

次に例を示します。

```
     .
     .
     .
     .DATA
     GLOBAL   FinalCount:WORD,PromptString:BYTE
FinalCount   DW  ?
     .
     .
     .
     .CODE
     GLOBAL  DoReport:NEAR,TallyUp:FAR
TallyUp  PROC FAR
     .
     .
     .
     call DoReport
     .
     .
     .
```

この例では*FinalCount*と*TallyUp*が定義されており，これらのラベルはパブリックラベルになっています。一方*PromptString*と*DoReport*はモジュール内で定義されていないため，これらのラベルは外部ラベルとして使われます。他のモジュール内ではパブリックラベルとして使われる可能性があります。

GLOBAL疑似命令はインクルードファイル内で使うと便利です（インクルードファイル

の詳細は次の節で説明します)。マルチモジュールプログラム内のすべてのモジュールから参照させたいラベルは，インクルードファイル内でまとめて宣言しておき，このインクルードファイルをそれぞれのモジュール内に挿入するのがよいでしょう。しかし**PUBLIC**や**EXTRN**を使うことはできません。なぜなら**EXTRN**はラベルが定義されているモジュール内では宣言できず，**PUBLIC**はラベルが定義されているモジュール内で**のみ**宣言できるためです。一方**GLOBAL**はすべてのモジュール内で宣言できるため，グローバルにしたいすべてのラベルを宣言するインクルードファイルを作っておき，このファイルをすべてのモジュール内に入れておくことができます。

インクルードファイル

複数のソースモジュール内に同一のアセンブラソースコードのブロックを挿入したい場合があります。またプログラムの各所でイコール定義またはマクロ命令を共有したり，複数のプログラムでそれらを再利用したい場合があります。また複数のリンク可能なモジュール（たとえばROM内に格納されるプログラムなど）に分割したくはないが，1つのファイルとしておくには長すぎるプログラムを扱わなければならない場合があります。これらの問題は**INCLUDE**疑似命令を使うことですべて解決することができます。

Turbo Assemblerは**INCLUDE**疑似命令を見つけると，カレントモジュール内のその場所をマークし，ディスク上の指定されたインクルードファイルを読み込みます。次にインクルードファイルのアセンブルを開始します。この際インクルードファイル内の行は，あたかもカレントモジュール内にあらかじめ存在していたかのように処理されます。インクルードファイルの終わりに達すると，Turbo Assemblerはカレントモジュール内の**INCLUDE**疑似命令の直後の行に戻り，そこからアセンブルを再開します。ここでのキー ポイントは，インクルードファイルのテキストはカレントアセンブラモジュール内の**INCLUDE**疑似命令の位置にそのままの形で挿入されるということです。

次に例を示します。プログラムMAINPROG.ASMの内容は次のとおりです。

```
.
.
.
.CODE
mov   ax,1
INCLUDE   INCPROG.ASM
push ax
.
.
.
```

またINCPROG.ASMの内容は次のとおりです。

```
mov   bx,5
add   ax,bx
```

MAINPROG.ASMをアセンブルすると結果は次のようになります。

```
.
.
.
.CODE
mov  ax,1
mov  bx,5
add  ax,bx
push ax
.
.
.
```

インクルードファイルはネストすることができます。つまりインクルードファイル内に別のインクルードファイルを入れることができるのです。Turbo Assemblerはインクルード行の左端に番号を付けるため，リストファイル内のインクルード行を識別するのは容易です。この番号によってモジュールファイルのネストレベルを示します（インクルードファイルは任意のレベルまでネストすることができます)。

ではTurbo Assemblerはどのようにしてインクルードファイルの場所を見つけるのでしょう？ **INCLUDE**のオペランドにファイル名にドライブ名やパス名を付けた場合は，指定されたディレクトリが検索され，ファイル名のみを指定してドライブ名やパス名を付けない場合は，カレントディレクトリが検索されます。カレントディレクトリ内にファイルが見つからない場合は，/**I**コマンドラインスイッチで指定されたディレクトリが検索されます。たとえばコマンドラインが

```
TASM -ic:¥include testprog
```

で，TESTPROG.ASM内に

```
INCLUDE  MYMACROS.ASM
```

という行がある場合は，はじめにカレントディレクトリ内でMYMACROS.ASMが検索され，そこで見つからなければ次にディレクトリC：¥INCLUDE内でMYMACROS.ASMが検索されます。どのディレクトリ内でも見つからない場合はエラーとなります。

円記号¥（IBM PCではバックスラッシュ＼）を**INCLUDE**パス指定の中に使うことができます。これによってMASMとの互換性を保つことができます。

インクルードファイルを使えば，多くのアセンブラモジュールからマクロライブラリを参照することができます。インクルードファイルはまた，1つのプログラムの複数モジュール間でイコール定義，グローバルラベル宣言，データセグメントなどを共有する際に非常に役に立ちます。別々のソースモジュールを容易にリンクすることができるため，インクルードファイルをプログラムに対して使うことはほとんどありません。しかし必要に応じてインクルードファイル内にプログラム（つまり任意のアセンブリ命令）を入れても問題はありません。

リストファイル

Turbo Assemblerではアセンブルの結果として，通常1つのオブジェクトファイル(.OBJ)を作成します。オブジェクトファイルの名前はソースファイル(.ASM)と同じです。アセンブルの際は必要に応じてリストファイル（.LST）を作るように指定することができます。このためには，コマンドラインでカンマ（,）を2つ（またはファイル名を2つ）追加指定するだけです。たとえば次のコマンドラインはHELLO.ASMをアセンブルしオブジェクトファイルHELLO.OBJを作ります。

```
TASM hello
```

次のコマンドラインはリストファイルHELLO.LSTも作ります。

```
TASM hello,,
```

次のコマンドラインはどちらもリストファイルHELLO.LSTを作ります。

```
TASM hello,hello,hello
```

および

```
TASM /L hello
```

オブジェクトファイル名とリストファイル名は，どちらもソースファイル名と同じである必要はありません。しかしこれらのファイル名をソースファイル名とわざわざ違うものにする理由もありません。

基本的にはリストファイルは，アセンブル結果に関する多様な情報をソースファイルに付加したファイルです。リストファイルは1つ1つの命令に対応するマシンコードと各行のマシンコードのカレントセグメント内オフセットをリストします。さらにプログラム内で使われているラベル（の値と型）とセグメントに関する情報テーブルおよび各セグメントの属性もリストします。

またTurbo Assemblerでは，指定さえすればソースファイル内で使われているすべてのラベルに関するクロスリファレンステーブルを作ることもできます。このテーブルは各ラベルの定義場所とその参照箇所を示すものです(第3章の/Cコマンドラインオプションを参照してください)。

次にリストファイルの基本項目(アセンブルされたマシンコードと各命令のオフセット)を見てみましょう。

アセンブル情報付きのソースコード

次にHELLO.ASMのリストファイルの例を示します。

```
Turbo Assenbler Version 1.0  06-29-88 16:21:27                    Page 1

HELLO.ASM

1                       DOSSEG
2 0000                  .MODEL SMALL
3 0000                  .STACK 100h
4 0100                  .DATA
5 0000   48 65 6C 6C 6F 2C 20 + HelloMessage DB 'Hello, world',13,10,12
6        77 6F 72 6C 64 0D 0A +
7        0C
8       = 000F          HELLO_MESSAGE_LENGTH EQU $ - HelloMessage
9 000F                  .CODE
10 0000 B8 0000s        mov  ax,@data
11 0003 8E D8           mov  ds,ax                ; DSにデータセグメントをセット
12 0005 B4 40           mov  ah,40h               ; デバイス書き込みのDOSファンク
                                                  ; ション番号
13 0007 BB 0001         mov  bx,1                 ; 標準出力
14 000A B9 000F         mov  cx,HELLO_MESSAGE_LENGTH
                                                  ; プリント文字数
15 000D BA 0000r        mov  dx,OFFSET HelloMessage
                                                  ; プリント文字列
16 0010 CD 21           int  21h                  ; "Hello, world"をプリント
17 0012 B4 4C           mov  ah,4ch               ; プログラム終了のDOSファンク
                                                  ; ション番号
18 0014 CD 21           int  21h                  ; プログラム終了
19                      END

Turbo Assenbler Version 1.0  06-29-88 16:21:27                    Page 2
Symbol Table

Symbol Name             Type    Value

??DATE                  Text    "06-29-88"
??FILENAME              Text    "HELLO   "
??TIME                  Text    "16:21:26"
??VERSION               Number  004A
@code                   Text    _TEXT
@codesize               Text    0
@cpu                    Text    0101H
@curseg                 Text    _TEXT
@data                   Text    DGROUP
@datasize               Text    0
@filename               Text    HELLO
@wordsize               Text    2
HELLOMESSAGE            Text    DGROUP:0000
HELLO_MESSAGE_LENGTH    Number  000F
```

```
Group & Segments          Bit Size Align  Combine Class

DGROUP                    Group
  STACK                   16  0100  Para  Stack   STACK
  _DATA                   16  000F  Word  Public  DATA
  TEXT                    16  0016  Word  Public  CODE
```

リストファイルの各ページの先頭にはヘッダがプリントされます。ヘッダはTurbo Assemblerのバージョン，アセンブル日付と時刻，ページ番号から構成されます。

リストファイルはソースコード部分とシンボルテーブルの2つの部分から成り，ソースファイル名を含むヘッダとオリジナルのアセンブラコードおよびアセンブラのソースコードとそのマシンコードに関する情報を示します。アセンブル中に発生したエラーは該当行の直後に挿入されます。

リストファイル中のコード行は次の形式です。

```
<depth> <line number> <offset> <machine code> <source>
```

■ 〈*depth*〉はリストファイル中のインクルードファイルとマクロ命令のネストレベルを示します。

■ 〈*line number*〉はリストファイル中の行番号（ヘッダとタイトル行を除く）を示します。行番号は行参照のために使われ，クロスリファレンス機能を使うときは特に役に立ちます。リストファイルHELLO.LSTの行番号1が**DOSSEG**疑似命令，行番号2が.**MODEL**疑似命令です。

〈*line number*〉フィールド中の行番号はソースモジュールの行番号ではないことに注意してください。たとえばマクロ命令を展開したりファイルのインクルードを行なうと，たとえソースモジュール内のカレント行が同じでも行番号フィールドが増えます（たとえばクロスリファレンスリストにある）行番号からソースファイルを参照するためには，リストファイルの行番号から該当する命令を見つけ，次にソースファイルの同一行を（行番号ではなく目で見て）見つける必要があります。

■ 〈*offset*〉はアセンブラのソース行から作られたマシンコードの先頭のカレントセグメント内オフセットを示します。たとえば*HelloMessage*はデータセグメントのオフセット*0*から始まります。

■ 〈*machine code*〉は16進数のバイトまたはワードの並びで，ソースファイルからアセンブルされたものです。たとえば**MOV AX, @data**はコードセグメントのオフセット0から始まります。それぞれの命令のオフセットフィールドの右側にある情報はマシンコードです。たとえば**MOV AX, @data**のマシンコードはB8 0000s（すべて16進数）です。0B8hは機械語でAXレジスタに定数をロードする命令のマシンコードで，0000sはAXレジスタにロードされる dataの定数です（実際は0000sは**@data**の値を入れるた

めの受け皿です。後ですぐに値を入れます)。**MOV AX, @data**をアセンブルすると3バイトのマシンコードとなります。リストファイルでは**MOV AX, @data**の直後つまりコードセグメントのオフセット3の位置に**MOV DS, AX**があるのがわかります。これは**MOV AX, @data**がオフセット0から始まる3バイト命令であることから明らかなことです。**MOV DS, AX**をアセンブルして生成されるマシンコード8E D8は2バイトであるため，次の命令はオフセット5から始まっています。

■ 〈*source*〉はオリジナルのソースファイルの行です（コメントだけの行など）行によってはマシンコードが作られない行もあります。これらの行には〈offset〉フィールドや〈machine code〉フィールドはなく，行番号のみが付けられています。

すでに述べたように，**@data**に対する値0000sは，次の命令

```
mov   ax,@data
```

の中の実際の数値を入れるための受け皿に過ぎません。セグメントの値は，Turbo Assemblerではなくリンカによって割り当てられます。Turbo Assemblerはマシンコードの後に文字"s"を付けます。これはリンカによってリンク後に再配置されたセグメントの相対的なセグメント値が割り当てられ，さらにプログラムをメモリ上にロードしたとき，絶対値に直されます。

同様に次の命令

```
mov   dx,OFFSET HelloMessage
```

をアセンブルするとそのマシンコードの最後には文字"*r*"が付けられます。この文字は，後でリンカによってセグメントが連結されたときにオフセットが再配置されることを示します。

Turbo Assemblerのアセンブルリストで使われるこのようなアセンブルの特性文字の一覧を次にまとめます。

文字	意味
r	モジュール内シンボルのオフセットの値が後で入る。
s	モジュール内シンボルのセグメントの値が後で入る。
sr	モジュール内シンボルのセグメントとオフセットの値が後で入る。
e	外部シンボルのオフセットの値が後で入る。
se	外部シンボルのポインタの値が後で入る。
so	セグメントの値のみが後で入る。
+	切り捨てられたコード

オブジェクトコードリストにおいて文字*"r", "s", "sr"*は，モジュール内シンボルのオフセット，セグメント，ポインタ（セグメントとオフセット）型をそれぞれ示します。文字*"e"*は外部シンボルのオフセット型，*"se"*は外部シンボルのポインタ型を示します。外部シンボルのセグメント型はローカルシンボルと同様文字*"s"*で表わされます。オブジェクトコードフィールドの最後に文字"＋"が付いている場合は，リストの都合で切り捨てられたコードがあることを示します。

リストの左端のフィールドは**レベルカウンタ**です。このフィールドはメインファイルのアセンブル時にはブランクになっていますが，インクルードファイルを呼び出すとインクルードレベルによって1, 2, 3, ...と変化します。同様にマクロ命令の展開でもレベルカウンタとして機能します。

リストファイル中にはバイト（2桁の16進数）とワードのマシンコードがあります。これには論理的な規則があります。つまり**OFFSET** *HelloMessage*（16ビットのオフセット）などのワードをアセンブルしたときはそのマシンコードは常にワードで表わされます。このワード格納形式はたいへん便利です。従来の8086で使われている「下位バイトが先で上位バイトが後」のワード格納形式は，バイトの順序が逆で見にくいものでした。

次に例を示します。

```
mov  ax,1234h
```

この例をアセンブルすると0B8h, 034h, 012Hの順番で3バイトのマシンコードが生成します。もしこのマシンコードを従来の8086のワード格納形式でストアすると次の順番になります。

```
B8 34 12
```

Turbo Assemblerのリストでは次のようになり，非常に見やすくなります。

```
B8 1234
```

このマニュアルで<*offset*>フィールドを説明するときは，プログラム中のラベルや行のカレントセグメント内オフセットについて言及しています。では，あるラベルや行がどのセグメントに属しているのかはどうしたらわかるのでしょうか？　それは次に説明するシンボルテーブルの役割です。

シンボルテーブルのリスト

リストファイルの後半は"Symbol Table"というヘッダで始まり，2つのテーブルがあります。1つはプログラム内で使われているラベルのリストで，他方はセグメントのリストです。

ところで，リストファイルのシンボルテーブル部分が必要なければ，**/N**コマンドラインスイッチを指定します。そうすればソースコードの部分のみリストされます。

ラベルテーブル

リストの第1テーブル（ラベルテーブル）は，ソースコード内のすべてのラベルの一覧を（その型と値とともに）アルファベット順で示します。たとえばリストファイルHELLO.LST内には次の行があります。

```
HELLOMESSAGE    BYTE DGOUP:0000
```

*HELLOMESSAGE*はラベル名つまりシンボルです。Turbo Assemblerは，**/MX**または**/ML**のコマンドラインスイッチを指定されない限りすべてのシンボルを大文字に変換するので，HELLOMESSAGEも大文字になっています。**BYTE**は*HelloMessage*で参照されるデータ要素のデータサイズを表わします。**DGROUP:0000**はラベル*HelloMessage*の値で，このラベルによって参照される文字列がセグメントグループDGROUP内のオフセット0から始まることを示します（前の節では，*HelloMessage*のマシンコードは文字"*r*"が付いていたことを思い出してください。これは，**DGROUP**内の他のセグメントがプログラムにリンクされると，*HelloMessage*は別のオフセットに再配置される可能性があることを表わしています。セグメント再配置に関する情報はリンカが出力するマップファイルにリストされます）。

同様に*ProgramStart*は，_TEXT:0000の値を持ったnear型のラベルとして表わされています。_TEXTは，.CODEで定義されたセグメントの名前です。したがって*ProgramStart*がコードセグメントの先頭アドレスです。これでラベルがどのセグメントに属しているかを

知る方法がわかりました。ラベル一覧の"value"フィールドにそのラベルが属しているセグメント情報が示されます。

リストファイルHELLO.LST内の他のラベルは，簡略セグメント疑似命令が使われたときにTurbo Assemblerによってあらかじめ定義されたものです。これらのラベルはすべて文字列に割り当てられ，"value"として_**TEXT**や**DGROUP**などの値を持っています。

ラベルには次の型があります。

ABS	**DWORD**	**NUMBER**	**TBYTE**
ALIAS	**FAR**	**QWORD**	**TEXT**
BYTE	**NEAR**	**STRUCT**	**WORD**

この章の最初でも述べたように，イコール定義ラベルは定数または文字列に割り当てられたラベルです。ラベルテーブルの**value**フィールドはそれらの値を示します。***HelloMessage***などのメモリアドレスに関連したラベルについては，valueフィールドにそのアドレスが示されます。

ラベルテーブルは，ソースコード内で使われているすべてのラベルの型と値の一覧表ということができます。

グループとセグメントのテーブル

リストのシンボルテーブル部のもう1つのテーブルは，グループとセグメントのテーブルです。**DGROUP**などのセグメントグループは単にGroupとして示されています。これはセグメントグループがそれ自身の属性を持っておらず，1つまたは2つ以上のグループから構成されるためです。グループとセグメントのテーブル内では，グループ名の直後に2桁インデント（字下げ）してセグメントを示すことによって，モジュール内でのグループとそのメンバ(構成セグメント)の関係を示します。HELLO.LSTでは，**STACK**セグメントと_**DATA**セグメントが，セグメントグループ**DGROUP**のメンバとなっています。

セグメントは属性を持っています。またグループとセグメントのテーブルには各セグメントの属性が示されます。このテーブルは各セグメントに関する情報（左からデータサイズ，セグメントサイズ，アラインメント，結合型，セグメントクラス）を示します。次に各項目について説明します。

データサイズ：

常に16です。ただし80386プロセッサ用のアセンブルコード内の**USE32**セグメントに関しては例外です（**USE32**セグメントについては第10章を参照してください)。

セグメントサイズ：

4桁の16進数で表わされます。たとえば**STACK**セグメントのサイズは0200h(10進の512)バイトです。

アラインメント：

セグメントの開始メモリアドレスの単位を示します。次にアラインメントの型を示します。

BYTE	セグメントは任意のアドレスから開始できる。
DWORD	セグメントは4の倍数のアドレスから開始できる。
PAGE	セグメントは256の倍数のアドレスから開始できる。
PARA	セグメントは16の倍数のアドレスから開始できる。
WORD	セグメントは偶数アドレスから開始できる。

リストファイルHELLO.LSTでは，**STACK**セグメントは，パラグラフ(Para)単位で開始しています。また**_DATA**セグメントと**_TEXT**セグメントは，ワード(Word)単位で開始しています（アラインメントについては第10章を参照してください)。

結合型：

同一名のセグメントがどのように他のセグメントと結合しているかを示します。たとえば**PUBLIC**結合型の複数同一名セグメントを結合して1つの大きなセグメントにしたり，**COMMON**結合型の複数セグメントをオーバレイして共通セグメントにすることができます（結合型については第10章を参照してください)。

セグメントクラス：

セグメントが属するクラス（**CODE, DATA, STACK**など）を示します。リンカはこの情報を使ってセグメントの順番を決め，複数セグメントをリンクして1つのプログラムにします（セグメントクラスについては第10章を参照してください)。

クロスリファレンステーブル

リストファイルのシンボルテーブル部からラベル，グループ，セグメントに関するほとんどの情報が得られます。しかしラベル，グループ，セグメントの定義場所と使用場所は提示されません。すなわちシンボルテーブルはラベル，グループ，セグメントのクロスリ

ファレンスを示すものではありません。クロスリファレンスシンボルの情報があれば，プログラムデバッグの際のラベル検索やプログラム追跡が容易になります。

クロスリファレンス情報のリスト出力を指定するには2通りの方法があります。1つは次のように/Cコマンドラインスイッチを指定します。

```
TASM /c hello,,
```

この例ではリストファイルHELLO.LSTの中にクロスリファレンス情報を出力します。ただし/Cだけでは不十分で，クロスリファレンスの出力先のリストファイル名を指定しなければなりません。 次のようにコマンドラインで4番目のフィールドを付け加える方法もあります。

```
TASM hello,hello,hello,hello
```

または

```
TASM hello,,,
```

第2章の2番目のサンプルプログラムREVERSE.ASMをアセンブルする場合は/Cスイッチを使います。

```
TASM /C reverse,,
```

この場合は次のリストファイルREVERSE.LSTが作られます。

```
REVERSE.ASM

1                             DOSSEG
2                             .MODEL  SMALL
3                             .STACK  200h
4                             .DATA
5        = 03EB         MAXIMUM_STRING_LENGTH  EQU  1000
6 0000   03E8*(??)      StringToReverse  DB  MAXIMUM_STRING_LENGTH DUP(?)
7 03E8   03E8*(??)      ReverseString    DB  MAXIMUM_STRING_LENGTH DUP(?)
8                             .CODE
9                       ProgramStart:
10 0000 B8 0000s              mov  ax,@data
11 0003 8E D8                 mov  ds,ax        ; DSをデータセグメントのポイントにセット
13 0005 B4 3F                 mov  ah,3fh       ; ハンドラからの読み込みDOSファンクショ
                                                ; ン番号
14 0007 BB 0000               mov  bx,0        ; 標準入力を指定
15 000A B9 03E8               mov  cx,MAXIMUM_STRING_LENGTH
16                                              ; 最大文字数まで読み込み
17 000D BA 0000r              mov  dx,OFFSET StringToReverse
18                                              ; プリント文字列をストア
19 0010 CD 21                 int  21h          ; 文字列読み込み
```

```
20 0012 23 C0            and  ax,ax        ; 文字が読み込まれたか?
21 0014 74 1F            jz   Done         ; no, 読み込み終了
22 0016 8B C8            mov  cx,ax        ; cxに文字列の長さをセット
23                                         ; (カウンタ)
24 0018 51               push cx           ; 文字列の長さをセーブ
25 0019 BB 0000r         mov  bx,OFFSET StringToReverse
26 001C BE 03E8r         mov  si,OFFSET ReverseString
27 001F 03 F1            add  si,cx
28 0021 4E               dec  si           ; 逆順文字列バッファの
29                                         ; 終わりをポイント
30                    ReverseLoop:
31 0022 8A 07            mov  al,[bx]      ; 次の文字を読み込み
32 0024 88 04            mov  [si],al      ; 逆順に文字をストア
33 0026 43               inc  bx           ; 次の文字をポイント
34 0027 4E               dec  si           ; 直前の場所をポイント
35                                         ; buffer
36 0028 E2 F8            loop ReverseLoop;次の文字を移動
37 002A 59               pop  cx           ; 文字列の長さを復帰
38 002B B4 40            mov  ah,40h       ; ハンドラへの書き込みDOSファンクション
                                           ; 番号
39 002D BB 0001          mov  bx,1         ; 標準出力を指定
40 0030 BA 03E8r         mov  dx,OFFSET ReverseString
                                           ; 文字列をプリント
41 0033 CD 21            int  21h          ; 逆順文字列をプリント
42                    Done:
43 0035 B4 4C            mov  ah,4ch       ; プログラム終了のDOSファンクション
                                           ; 番号
44 0037 CD 21            int  21h          ; プログラム終了

45                       END  ProgramStart

Symbol Table

Symbol Name              Type     Value                Cref defined at #

@code                    Text     _TEXT                #2 #8
@curseg                  Text     _TEXT                #2 #3 #4  #8
DONE                     Near     _TEXT:0035           21 #42
MAXIMUM_STRING_LENGTH    Number   03E8                 #5  6  7  15
PROGRAMSTART             Near     _TEXT:0000           #9  45
REVERSWELOOP             Near     _TEXT:0022           #30 36
REVERSESTRING            Byte     DGROUP:03E8          #7  26 40
STRINGTOREVERSE          Byte     DGROUP:0000          #6  17 25

Group & Segments         Bit Size Align Combine Class  Cref defined at #

DGROUP                   Group                         #2  2 10
  STACK                  16  0200 Para  Stack   STACK  #3
  _DATA                  16  07D0 Word  Public  DATA   #2  #4
_TEXT                    16  0039 Word  Public  CODE   #2  2 #8  8
```

リストファイルには，アセンブル情報コード付きのソースコードとシンボルテーブルにリストされます。この他シンボルテーブルにはクロスリファレンスフィールドがあります。このフィールドは各シンボル（ラベル，グループ，セグメント）が参照されたすべての

プログラム内の行番号を表わします。シンボルが定義された行番号の前には"#"が付けられます。

たとえばこのサンプルプログラムで，*MAXIMUM_STRING_LENGTH*というラベルの定義場所と参照場所（使用場所）を見つけてみましょう。リストによれば，そのラベルは行番号5で定義されています。またリストの先頭部分を見ればそのことが確認できます(ラベルテーブルからこのラベル*MAXIMUM_STRING_LENGTH*は数値型で，その値が03E8hつまり10進数の1000であることがわかります)。

*MAXIMUM_STRING_LENGTH*のクロスリファレンスフィールドから，このラベルは行番号6, 7, 15で使われていることがわかります。リストの先頭を見ればそのことが確認できます。

/Cスイッチを使うとファイル全体に関するクロスリファレンスリストが作られます。大きなソースモジュールでクロスリファレンステーブルを作ると，リストのサイズが非常に大きくなってしまいます。したがって，ときにはクロスリファレンスリストが必要でない場合もあります。Turbo Assemblerには，リストの任意の範囲でクロスリファレンスをリストするか否かを指定する疑似命令があります。

%CREF疑似命令は以降の行のクロスリファレンスリストを作ります。一方**%NOCREF**疑似命令は以降の行のクロスリファレンスリストを作りません。これらの疑似命令はいずれもコマンドラインの**/C**スイッチに優先します。クロスリファレンスがソースモジュール全体で指定されている場合は，シンボルテーブルにすべてのラベル，グループ，セグメントの定義行番号が示されます。またクロスリファレンスが特定の行範囲に対して指定されている場合は，その範囲内でラベル，グループ，セグメントを参照または使用している行番号がクロスリファレンスフィールドに示されます。

次に例を示します。

```
      .
      %NOCREF
ProgramStart  PROC                 ; 行1
      .
      .
      .
      jmp  LoopTop                 ; 行2
      .
      .
      .
      %CREF
LoopTop:                           ; 行3
      .
      .
      .
      loop LoopTop                 ; 行4
      %NOCREF
      mov  ax,OFFSET ProgramStart  ; 行5
```

この例では行1はクロスリファレンスが無効にされている範囲内にありますが，ラベル*ProgramStart*の定義行（#付きの番号）としてリストに出力されるはずです。なぜならばモジュール内のどこか一箇所でクロスリファレンスが指定にされていれば，すべてのラベルの定義行がリストに出力されるからです。同様に行3も*LoopTop*の定義行としてリストに出力されます。

行4は**%CREF**と**%NOCREF**の間に挟まれているため，*LoopTop*の参照行として示されます。しかし行2はクロスリファレンスが無効にされている範囲にあるため，*LoopTop*の参照行としてはリストに出力されません。同様に行5も*ProgramStart*の参照行としてリストに出力されません。

他のアセンブラとの互換性を保つために**.CREF**と**.XCREF**が備っています。これらの疑似命令はそれぞれ**%CREF**や**%NOCREF**と同様にクロスリファレンスをコントロールするものです。

リストの内容と形式の指定

Turbo Assemblerは，ソースコードリストの内容や形式を指定するためのリスト制御疑似命令を備えています。リスト制御疑似命令には，リスト行選択疑似命令（リストする情報つまりリストファイルに入れる情報を選択する命令）とリスト形式指定疑似命令（リストの形式を指定する命令）の2種類があります。

リスト行選択疑似命令

リスト行選択疑似命令は特定の行をリストするか否かを指定します。この疑似命令を使えば，リストから不必要な情報を一時的に消去してリストファイルを扱いやすいサイズにすることができます。

%LISTと%NOLIST

%LISTと**%NOLIST**は，リスト行選択疑似命令の中でもっとも基本的な疑似命令です。これらの疑似命令はリストファイル中の以降の行をリストするか否かを指定します。次に例を示します。

```
.
.
.
%NOLIST
mov  ax,1
%LIST
mov  bx,2
%NOLIST
add  ax,bx
.
.
.
```

この例では中程の行mov bx,2だけをリストに入れます。デフォルトでは**%LIST**が指定されています。

%CONDSと%NOCONDS

%CONDSと**%NOCONDS**は，偽条件と条件文をリストするか否かを指定します。条件文は通常リストしません。次に例を示します。

```
     .
     .
     .
     %CONDS
IFE IS8086
     shl  ax,7
ELSE
     mov  cl,7
     shl  ax,cl
ENDIF
     .
     .
     .
```

この例ではアセンブル時に真となる条件部分だけではなく，真と偽の両方の条件部分を条件アセンブル疑似命令とともにリストに入れます。

%INCLと%NOINCL

%INCLと**%NOINCL**は，**INCLUDE**疑似命令によって他のファイルからインクルードしたテキスト行をリストするか否かを指定します。インクルードテキスト行は通常はリストに入ります。次に例を示します。

```
.
.
.
%NOINCL
INCLUDE   HEADER.ASM
%INCL
INCLUDE   INIT.ASM
.
.
.
```

この例では,HEADER.ASMからインクルードした行はリストに入れませんが,INIT.ASMからインクルードした行はリストに入れます(ただし2つの**INCLUDE**疑似命令自身は両方ともリストに入れます)。

%MACSと%NOMACS

%MACSと**%NOMACS**は,マクロ命令の展開結果をリストするか否かを指定します。マクロ命令の展開結果は通常はリストに入りません。次に例を示します。

```
          .
          .
          .
MAKE_BYTE MACRO   VALUE
          DB   VALUE
          ENDM
          .
          .
          .
          %NOMACS
          MAKE_BYTE 1
          %MACS
          MAKE_BYTE 2
          .
          .
          .
```

この例では*MAKE_BYTE*マクロ命令の最初の拡張命令**DB 1**はリストに入れませんが,2番目の拡張命令**DB 2**はリストに入れます(ただし2つの**MACRO**疑似命令自身は両方ともリストに入れます)。

%CTLSと%NOCTLS

%CTLSと**%NOCTLS**は,リスト制御疑似命令をリストするか否かを指定します。リスト制御疑似命令は通常はリストしません。次に例を示します。

```
.
.
.
%NOCTLS
%NOINCL
%CTLS
%NOMACS
.
.
.
```

この例ではリスト制御疑似命令%NOINCLはリストしませんが，リスト制御疑似命令**%NOMACS**はリストします。

%UREFと**%NOUREF**は，参照されなかったシンボル(定義されただけで参照されなかったシンボル）をシンボルテーブルに入れるか否かを指定します。このようなシンボルは通常リストに入れます。この疑似命令のどちらかを使いたい場合は，クロスリファレンスリストの出力を指定しておく必要があります。

%SYMSと**%NOSYMS**は，リストファイルにシンボルテーブルを入れるか否かを指定します（この節のいくつかのリスト例からもわかるように）シンボルテーブルは通常リストファイルに入ります。

リスト形式指定疑似命令

リスト形式指定疑似命令はリストファイルの形式を変更します。これらの疑似命令を使えば，必要に応じてリストファイルの体裁を整えることができます。

%TITLE疑似命令はリストファイルのソースコード部の各ページの先頭にプリントするタイトルを指定します。**%SUBTTL**疑似命令はリストの各ページのタイトルの下のサブタイトルを指定します。プログラム内にはサブタイトルをいくつでも指定することができます。例としてSPACEWAR.ASMというモジュールの内容を次に示します。

```
.
.
.
%TITLE   'Space Wars Game Program'
%SUBTTL  'Gravitational Effects Subroutines'
.
.
.
```

この例ではソースコードの各ページには次のようなタイトルが付けられます。

```
Turbo Assembler Version 1.0  10 Feb 88   06-27-88  21:53:35   Page 1
SPACEWAR.ASM
Space Wars Game Program
Gravitational Effects Subroutines
```

%NEWPAGE疑似命令は，リストを強制的に改ページします。

%TRUNC疑似命令は，リストの最大幅を越えたフィールドを切り捨てます。**%NOTRUNC**疑似命令は最大幅を越えた分を次の行に送ります。通常は次の行に送ります(**%NOTRUNC**がデフォルトです)。

%PAGESIZE疑似命令は，132桁 x 66行のページを作成します。この疑似命令はプリンタに対してページサイズコマンドを直接送るものではありません。リストファイルをプリントする前に，プリンタのセットアップを行なってください。その後**%PAGESIZE**を使ってそれに合ったリストのページサイズを指定してください。

フィールド幅指定疑似命令

リストのソースコード部の5つのフィールド幅を指定する疑似命令があります。ソースコード部の形式は次のとおりです。

```
<depth> <line number> <offset> <machine code> <source>
```

<*depth*>フィールド以外の4つのフィールドについては，すでに解説しました。<*depth*>フィールドはカレント行のマクロ命令またはインクルードのネストレベル(ネストの深さ)を表わします。たとえばカレント行がマクロ命令で生成され，そのマクロ命令がさらに他のマクロ命令から呼び出されている場合，ネストレベルは2となります。次にフィールド幅指定疑似命令を示します（どの疑似命令でもフィールド幅は文字数で指定します)。

%DEPTH <*depth*>フィールドの幅を指定します。
%LINUM <*line number*>フィールドの幅を指定します。
%PCNT <*offset*>フィールドの幅を指定します(このフィールドをプログラムカウンタのフィールドと考えれば，**%PCNT**は憶えやすくなります)。
%BIN <*machine code*>フィールドの幅を指定します。
%TEXT <*source*>フィールドの幅を指定します。

%PUSHLCTLと%POPLCTL

現在のリスト制御を一時的に変更したい場合があります。データテーブルの全データを

リストするためには，ラッピング（行送り）の指定とフィールド幅の調整を行なう必要があります。またデバックのためにすべての行の型をリストしたい場合もあります。リスト制御をいったん変更すると，元の状態に戻すことは非常にめんどうになります。なぜならリスト制御の設定はインクルードファイル内で，またはソースモジュールと無関係な場所で設定されていることがよくあるからです。

このようなときのために，*Turbo Assembler*は疑似命令**%PUSHLCTL**と**%POPLCTL**を備えています。**%PUSHLCTL**疑似命令は，現在のリスト制御状態を内部スタックに退避します。**%POPLCTL**疑似命令は，内部スタックから現在のリスト制御状態を復帰します(どちらの疑似命令も**16**レベルまでネストすることができます)。これらの疑似命令を使って退避・復帰できるリスト制御状態は，（**%TRUNC**や**%NOTRUNC**などの）疑似命令によって有効/無効を切り換えられるものだけです。引数に数値を指定する疑似命令(**%BIN**など)に関連するものは退避・復帰できません。次のプログラム例は，%PUSHLCTL疑似命令によって退避したリスト制御状態を**%POPLCTL**疑似命令によって復帰しています。つまり**%PUSHLCTL**の前と**%POPLCTL**の後でリスト制御状態が同じになっています。

```
.
.
.
%LIST
%TRUNC
%PUSHLCTL
%NOLIST
%NOTRUNC
%NEWPAGE
.
.
.
%POPLCTL
.
.
.
```

他のリスト制御疑似命令

他のアセンブラとの互換性を保つために，Turbo　Assemblerにはこの他にも**TITLE, SUBTTL, PAGE, .LST, .XLIST, LFCOND, SFCOND, TFCOND, .LALL, .SALL, .XALL**などのリスト制御疑似命令があります(これらの疑似命令の詳細については，リファレンスガイドの第2章を参照してください)。

アセンブル中のメッセージ表示

Turbo Assemblerにはアセンブル中にコンソールに文字列を表示する疑似命令があります。これらの疑似命令を使えば，アセンブル処理の進み具合いやソースの特定部分のアセンブルの終了を知ることができます。

DISPLAY疑似命令はアポストロフィ(')で囲まれた文字列を画面に表示します。**%OUT**疑似命令も画面に文字列を表示します(ただし文字列はアポストロフィで囲みません)。どちらの疑似命令も機能的には同じです。次に例を示します。

```
.
.
.
DISPLAY  'This message produced by DISPLAY'
%OUT     This message produced by %OUT
.
.
.
```

この例では次のメッセージを画面に表示します。

```
This message produced by DISPLAY
This message produced by %OUT
```

ソースコードの条件アセンブル

プログラムのソースモジュールのアセンブルの際に，バージョンを指定してアセンブルできたら非常に便利な場合があります。たとえばあるプログラムに対して8086の標準命令を使うバージョンと80186と80286の強力な命令を使うバージョンの2つのバージョンを作りたい場合などです。別々のソースモジュールを作って2つのバージョンとすればよいのですが，両方とも管理するのはたいへんです。もっとも簡単な解決方法は2つのバージョンを1つのソースモジュールにまとめてしまうことです。この場合はイコール定義ラベルを使って，アセンブル時にどちらのバージョンをアセンブルするかを指定します。

Turbo Assemblerにはこのような条件アセンブルのための疑似命令があります。次に例を示します。

```
     .
     .
     .
IF IS8086
     mov  ax,3dah
     push ax
ELSE
     push 3dah
ENDIF
     call GetAdapterStatus
     .
     .
     .
```

この例はラベル*IS8086*の値が0でない場合は値3dahを（8086に必要な2ステッププロセスによって）スタックにプッシュし，0の場合は80186と80286で使える**PUSH**の特殊形式（8086では使えません）によって値を直接スタックにプッシュします。また条件アセンブルを使うことによって，1つのプログラムで2つのバージョン（8086用のバージョンと80186/80286用のバージョン）をサポートすることができるようになります。

Turbo Assemblerは多様な条件アセンブル疑似命令を備えており，さまざまな方法でアセンブルエラーを見つけることが可能です。次に条件アセンブル疑似命令を見てみましょう。

条件アセンブル疑似命令

もっとも単純でしかも便利な条件アセンブル疑似命令は**IF**と**IFE**です。これらは**ENDIF**（または**ELSE**）と組み合わせて使われます。**IFDEF**と**IFNDEF**もよく使われます。**IFB, IFNB, IFIDN, IFDIF, IF1, IF2**なども場合によっては便利な疑似命令です。

IFとIFE

IFはオペランドの値が0でない場合にのみ，直後のコードブロック（**ELSE**または**ENDIF**までのコード範囲）をアセンブルします。オペランドは定数でも定数として評価できる式でもかまいません。次に例を示します。

```
      .
      .
      .
IF REPORT_ASSEMBLY_STATUS
     DISPLAY  'Reached assembly checkpoint 1'
ENDIF
      .
      .
      .
```

この**IF**条件は*REPORT_ASSEMBLY_STATUS*の値が0でない場合にのみ真となり，次のメッセージを表示します。

```
Reached assembly checkpoint 1
```

IF条件は**ENDIF**または**ELSE**で終了します。**IF**条件に**ELSE**が続く場合は，**IF**のオペランドが0のときのみ**ELSE**に続くコードブロックがアセンブルされます。**ELSE**に続くコードブロックは**ENDIF**で終了しなければなりません。**IF**条件はネスト可能です。次にその例を示します。

```
      .
      .
      .
; 配列が定義されているかチェックする
;（なければダイナミックに配置される）
IF DEFINE_ARRAY
; 配列が大きすぎないことを確認する
     IF (ARRAY_LENGTH GT MAX_ARRAY_LENGTH)
ARRAY_LENGTH  =  MAX_ARRAY_LENGTH
     ENDIF
; 指定があれば配列の初期値をセットする
     IF INITIALIZE_ARRAY
Array    DB  ARRAY_LENGTH DUP (INITIAL_ARRAY_VALUE)
     ELSE
Array    DB  ARRAY_LENGTH DUP (?)
     ENDIF
ENDIF
      .
      .
      .
```

この例ではIF条件の中でIFとIF...ELSEをネストしています。

IFEはオペランドが0の場合に以降のコードをアセンブルする点を除いて，**IF**と同じです。次の例では**IFE**疑似命令に続くソースコードは常にアセンブルされます。

```
      .
      .
      .
IFE  0
      .
      .
      .
ENDIF
      .
      .
      .
```

IFと同様，**IFE**にも**ELSE**疑似命令を使うことができます。

条件アセンブル疑似命令は実行時ではなくアセンブル時にのみ働きます。これらの疑似命令は，実行時の条件によって実行するコードを変更するC言語の**if**文とは異なり，アセンブル時の条件によって**アセンブル**するコードを変更します(つまりC言語の#if指令などと同じ働きをします)。

IFDEFとIFNDEF

IFや**IFE**は複数のバージョンにわけてプログラムを作る際の基本的な疑似命令です。関連する疑似命令として**IFDEF**と**IFNDEF**があります。

IFDEF疑似命令とそれに対応する**ENDIF**にはさまれたコードブロックは，**IFDEF**のオペランドとして指定されたラベルが存在するときのみ（つまり**IFDEF**疑似命令が実行された時点でラベルが定義済みの場合にのみ）アセンブルされます。次に例を示します。

```
      .
      .
      .
DEFINED_LABEL   EQU 0
      .
      .
      .
IFDEF DEFINED_LABEL
      DB    0
ENDIF
      .
      .
      .
```

この例では**DB**疑似命令はアセンブルされます。しかし*DEFINED_LABEL*のイコール定義を削除した場合(しかも*DEFINED_LABEL*がプログラム内のどこにも定義されていない場合）は，**DB**疑似命令はアセンブルされません。*DEFINED_LABEL*に割り当てられた値は**IFDEF**の働きに関係しません。

IFNDEFは**IFDEF**の反対です。つまりオペランドのラベルが定義されていない場合にのみ，後続のコードをアセンブルします。

IFDEFと**IFNDEF**はいったい何の役に立つのでしょうか。1つの用途は，複雑なプログラム内で**EQU**による同じラベルの二重定義を避けることです。**IFDEF**を使うと，定義しようとしたラベルがすでに定義されている場合はエラーとなり，二重定義を防ぐことができます。他の用途として，先のIF疑似命令と同様にアセンブルするプログラムのバージョンを選択することです。*INITIALIZE_ARRAYS*が0か否かをチェックするかわりに，あるラベルが定義されているか否かをチェックします。

Turbo Assemblerには，プログラムバージョンの簡便な選択方法として/**D**コマンドラインスイッチがあります。/**D**は関連ラベルを指定し，必要ならそのラベルに値を割り当てます。次にコマンドラインの例を示します。

```
TASM /dINITIALIZE_ARRAYS=1 test
```

この例では*INITIALIZE_ARRAYS*に1を割り当て，プログラムTEST.ASMをアセンブルします。

/**D**スイッチは便利ですが問題もあります。*INITIALIZE_ARRAYS*はコマンドライン上で指定されるものとしてプログラムが組まれている場合に，/**D**スイッチをタイプし忘れてしまうかもしれません。また，（特殊なケースとして）配列を初期化したいが/*dINITIALIZE*

_*ARRAYS*をタイプするのが面倒だと思うことがあるでしょう。

IFNDEF疑似命令を使えばこれらの問題を解決できます。**IFNDEF**を使ってラベル*INITIALIZE_ARRAYS*が（コマンドラインで）定義済みか否かをチェックし，ラベルが定義されていない場合のみ初期化が行なわれるようにすることができます。このようにコマンドラインからの指定の方が優先されますが，コマンドラインで指定しない場合はラベルのデフォルト値が決まっています。次の例は，ラベル*INITIALIZE_ARRAYS*が定義されていないことを確かめた後，そのラベルのイコール定義を行なうプログラムです。

```
      .
      .
      .
IFNDEF    INITIALIZE_ARRAYS
INITIALIZE_ARRAYS    EQU   0      ; デフォルト(初期化しない)
ENDIF
      .
      .
      .
```

この例のように**IFNDEF**疑似命令に対して未定義シンボルを使うと，パス依存型である旨の警告メッセージが出力されます。このメッセージは**IFNDEF**の条件ブロック内でシンボルを定義している限り無視することができます。Turbo Assemblerが1パスアセンブラであるため，機械命令や疑似命令がブロック内に入っているか否かを予想することができません。したがってこのようなメッセージが出力されるのです。

他の条件アセンブル疑似命令

IFB, IFNB, IFIDN, IFDIF疑似命令はマクロに渡されるパラメータをテストするものです（マクロについては第10章「Turbo Assemblerの応用プログラミング」で解説します)。**IFB**はそのオペランドであるマクロパラメータがブランクの場合に，後続のコードをアセンブルします。**IFNB**はパラメータがブランクでない場合に，後続のコードをアセンブルします。**IFB**と**IFNBI**はマクロパラメータ用の**FNDEF**や**IFDEF**ということができます。

次に例を示します。次のプログラムはマクロ命令*TEST*の定義例です。

```
; バイトまたはワードを定義するマクロ
;
; 入力：
;     VALUE = バイトまたはワードの値
;     DEFINE_WORD = 1: ワード定義, 0: バイト定義
;
; 注意: PARM2が指定されていないときはバイトを定義する
;
TEST MACRO    VALUE, DEFINE_WORD
IFB  <DEFINE_WORD>
     DB   VALUE                   ; パラメータ2が空白のときはバイトを定義
ELSE
  IF DEFINE_WORD
     DW   VALUE                   ; パラメータ2が0でないときはワードを定義
  ELSE
     DB   VALUE                   ; パラメータ2が0のときはバイトを定義
  ENDIF
ENDIF
     ENDM
```

この例では*TEST*が

```
TEST 19
```

で呼び出されている場合は19という値を持つバイトが定義され，*TEST*が

```
TEST 19,1
```

で呼び出されている場合は19という値を持つワードが定義されます。

IFIDNはオペランドの2つのマクロパラメータの値が同じ場合に，後続のコードをアセンブルします。IFDIFは2つの値が異なる場合に，後続のコードをアセンブルします。下に例は符号付きバイトをAXの符号付きワードに変換するマクロです。このマクロはソースオペランドがALのときはソースオペランドをALに転送しません。

```
;
; 8ビットレジスタまたはラベルのついたメモリアドレスの符号付きバイトを
; AXの符号付きワードに変換するマクロ命令
;
; 入力：
;     SIGNED_BYTE = 符号付きバイトがあるレジスタ名またはメモリアドレス
;
MAKE_SIGNED_WORD     MACRO     SIGNED_BYTE
IFDIFI    <AL>,<SIGNED_BYTE>             ; オペランドがALでないことをチェック
     mov   al,SIGNED_BYTE
ENDIF
     cbw
     ENDM
```

IFIDNと**IFDIF**は引数の大文字小文字の区別をします。これらの関連疑似命令**IFIDNI**

と**IFDIFI**は大文字区別の区別をしません。

IFB, IFNB, IFIDN, IFDIFのオペランドは不等号カッコ（<>）で囲んでください。

IF1と**IF2**はそれぞれアセンブルのパス1とパス2の有効性をチェックします。Turbo Assemblerは1パスアセンブラなのでこのような疑似命令があるのは妙ですが，実は2パス方式の他のアセンブラとの互換性を保つために用意されているのです。Turbo Assemblerでは，通常2パス方式のアセンブラ依存型プログラムは正しくアセンブルできないことを憶えておいてください。

疑似命令のELSEIFファミリ

疑似命令のIFファミリ(**IF, IFB, IFIDN**など)は**ELSEIF**ファミリ(**ELSEIF, ELSEIFB, ELSEIFIDN**など)に対応しています。これらの疑似命令はIF疑似命令のどれかと**ELSE**疑似命令の組み合わせと同様の働きをします。複数の条件や値をテストしたいとき，または特定のコードブロックのみをアセンブルしたいときに，これらの疑似命令を使えばプログラムが一層読みやすくなります。次にプログラム例を示します。

```
IF BUFLENGTH GT 1000
        CALL DOBIGBUF
ELSE
        IF BUFLENGTH GT 100
                CALL MEDIUMBUF
        ELSE
                IF BUFLENGTH GT 10
                         CALL SMALLBUF
                ELSE
                         CALL TINYBUF
                ENDIF
        ENDIF
ENDIF
```

このコードは次のように**ELSEIF**疑似命令を使えば一層読みやすくなります。

```
IF BUFLENGTH GT 1000
        CALL DOBIGBUF
ELSEIF BUFLENGTH GT 100
        CALL MEDIUMBUF
ELSEIF BUFLENGTH GT 10
        CALL SMALLBUF
ELSE
        CALL TINYBUF
ENDIF
```

この疑似命令はPascalやC言語の**case**文や**switch**文に通じるところがあります。しかし条件コードブロック内で**ELSEIF**テストを何度も繰り返し使う必要がなく，この疑似命令の方

がはるかに一般的です。次のプログラム例も有効です。

```
PUSHREG MACRO ARG
     IFIDN <ARG>,<INDEX>
            PUSH SI
            PUSH DI
     ELSEIFB <ARG>
            PUSH AX
     ENDIF
     ENDM
```

条件エラー疑似命令

Turbo Assemblerは条件エラー疑似命令を備えており，無条件にまたは条件によってアセンブルエラーを発生させることができます。条件エラー疑似命令には**.ERR, .ERR1, .ERR2, .ERRDEF, .ERRNDEF, .ERRB, .ERRNB, .ERRIDN, .ERRDIF, .ERRE, .ERRNZ, .ERRDIFI, .ERRIDNI**などがあります。ではなぜアセンブルエラーを意図的に発生させる必要があるのでしょうか。その理由は条件エラー疑似命令によって，サイズの不正なイコール定義ラベル，未定義ラベル，マクロパラメータの欠如などのプログラム内のさまざまな誤りを見つけることができるからです。

条件エラー疑似命令の一覧をもう一度見てください。条件エラー疑似命令は条件アセンブル疑似命令に非常によく似ていることがわかります。両者の違いは，条件エラー疑似命令のほとんどは同じ条件をテストすることです。たとえば**.ERRNDEF**はオペランドラベルが定義されていないときにエラーを発生します。これは**IFNDEF**がオペランドラベルが定義されていないときに，後続のコードをアセンブルするのと似ています。

.ERR, .ERR1, .ERR2

Turbo Assemblerは**.ERR**疑似命令を見つけると無条件にエラーを発生します。これらの疑似命令は単独ではあまり便利なものとは思えません。しかし条件アセンブル疑似命令と組み合わせて使うと威力を発揮します。

たとえば配列サイズのイコール定義に割り当てた値が最大値より大きいときに，エラーを発生させたい場合があります。次のそのプログラム例を示します。

```
IF (ARRAY_LENGTH GT MAX_ARRAY_LENGTH)
     .ERR
ENDIF
```

配列サイズが最大値以下のときは，Turbo Assemblerは**IF**ブロック内のコードをアセン

ブルしません。したがって**.ERR**はアセンブルされずエラーは起きません。

.ERR1と**.ERR2**の機能は，それぞれパス1とパス2における.ERRと同じです。Turbo Assemblerは1パスアセンブラです。これらの疑似命令は2パス方式の他のアセンブラとの互換性を保つために必要です。Turbo Assemblerでは，2パス方式のアセンブラ依存型プログラムは正しくアセンブルすることができないことを憶えておいてください。

.ERREと.ERRNZ

.ERRE疑似命令はオペランド(定数として評価されます)が0のときにエラーを発生します。**.ERRE**は**.IFE**と**.ERR**を組み合わせた機能を持っています。たとえば

```
.ERRE    TEST_LABEL-1
```

は

```
IFE  TEST_LABEL-1
    .ERRE
ENDIF
```

と同じです。

.ERREを使えば，関係式が偽を返したときにエラーを発生させることができます。なぜなら関係式が偽のときは結果として0を返すからです。

一方**.ERRNZ**はオペランドが0でないときにエラーを発生します。これは**IF**と**.ERR**を使うことと同じです。**.ERRNZ**を使えば関係式が真を返したときにエラーを発生させることができます。なぜなら関係式が真のときは結果として0以外の値を返すからです。次に例を示します。

```
.ERRNZ    ARRAY_LENGTH GT MAX_ARRAY_LENGTH
```

この例の働きは，前の節の例の**IF**と**.ERR**疑似命令と同じです。

.ERRDEFと.ERRNDEF

.ERRDEFはオペランドのラベルが定義されているときにエラーを発生します。一方**.ERRNDEF**はオペランドのラベルが定義されていないときにエラーを発生します。これらの疑似命令の働きは，**IFDEF**または**IFNDEF**と**.ERR**を組み合わせた場合と同じです。たとえば

```
.ERRNDEF  MAX_PATH_LENGTH
```

は

```
IFNDEF   MAX_PATH_LENGTH
     .ERR
ENDIF
```

と同じです。

他の条件エラー疑似命令

次に説明する4つの条件エラー疑似命令はマクロ命令内でのみ使われます。これらの疑似命令は，少し前の「他の条件アセンブル疑似命令」の節で述べたマクロ命令内で使われていた4つの条件アセンブル疑似命令に対応しています。

.ERRBはオペランドのマクロパラメータがブランクのときにエラーを発生します。一方**.ERRNB**はオペランドのマクロパラメータがブランクでないときにエラーを発生します。**.ERRIDN**はオペランドの2つのマクロパラメータが一致するときにエラーを発生します。一方**.ERRDIF**はオペランドの2つのマクロパラメータが一致しないときにエラーを発生します。

下にマクロ命令内にこれらの疑似命令を使った例を示します。このマクロ命令は2個以外のパラメータで呼び出されたときにエラーを発生します。**.ERRB**と**.ERRNB**によって*PARM2*がブランクでなく，*PARM3*がブランクであることが確認されます。次の**.ERRIDN**は*PARM2*がDXと等しくないことを確認します(この場合*PARM1*がロードされるとDXの内容は書き換えられます)。

```
;
; 定数,レジスタ,メモリアドレスのどれか2つの値を加え
; 結果をDXにストアするマクロ命令
;
; 入力:
;     PARM1 = 第1オペランド
;     PARM2 = 第2オペランド
;
ADD_TWO_OPERANDS     MACRO    PARM1,PARM2,PARM3
     .ERRB     <PARM2>              ; パラメータが2つあることをチェック
     .ERRNB    <PARM3>              ; パラメータが多すぎないかをチェック
     .ERRIDN   <PARM2>,<DX>         ; 第2パラメータとDXが等しくないことをチェック
     mov  dx,PARM1
     add  dx,PARM2
     ENDM
```

アセンブラプログラミングの落とし穴

どのプログラム言語もプログラミング上の特有の問題点をもっています。アセンブリ言語もその例外ではありません。以下にアセンブリ言語のプログラミングにおける誰もが陥りやすい落とし穴とそれを防ぐためのヒントを紹介します。

DOSに戻るのを忘れる

PascalやC言語ではプログラム中に明示的な終了コマンドが書かれていなくても，実行するコードがなくなるとプログラムは自動的に終了してDOSに戻ります。しかしアセンブリ言語ではそうはいきません。アセンブリ言語ではプログラマが明示的に記述したことしか実行しません。したがってDOSに戻るコマンドのないプログラムを実行すると，プログラムの終わりに達してもプログラムが停止せず，連続するメモリ内のコードを引き続き実行してしまいます。

次にプログラム例を示します。

```
     DOSSEG
     .MODEL   SMALL
     .CODE
DoNothing PROC NEAR
     nop
DoNothing ENDP
     END  DoNothing
```

これまでの説明を読む限り，CやPascalで}または**end.**を付けたようにアセンブラプログラムの最後に**ENDP**または**END**疑似命令を付けておけば，プログラムを正常終了できると思われるかもしれません。しかしそうではありません。上のプログラムをアセンブル，リンクすると**NOP**命令だけを持つ実行可能プログラムが作られます。アセンブラでは，(すべての疑似命令と同様)**ENDP**疑似命令はコードを生成しません。**ENDP**疑似命令は*DoNothing*のプロシージャの終了をアセンブラに指示するだけです。同様に**END**疑似命令は，このモジュールがこの位置で終了することとプログラムの開始位置が*DoNothing*であることを，アセンブラに対して知らせているに過ぎません。この例では，プログラムの実行が終了したときにDOSに制御を戻すための命令がどこにもありません。その結果，**NOP**命令以降のアドレスにどんな命令が入っているかわかりませんが，それを命令と解釈して実行してしまいます。こうなるともう絶望的です。コンピュータはハングアップして二度とDOSへ戻れ

なくなってしまいます。そうなったらシステムをリブートする以外にありません。

アセンブラプログラムからDOSへ戻る方法は1通りではありませんが，DOSファンクションの4Chを利用するのが一般的でしょう。次のプログラムは前の例をDOSファンクションを使って書き直したものです。

```
        DOSSEG
        .MODEL    SMALL
        .CODE
DoNothing PROC NEAR
        nop
        mov   ah,4Ch                       ; プログラム終了のDOSファンクション
        int   21h                          ; DOSファンクションを呼び出してプログラムを終了
DoNothing ENDP
        END   DoNothing
```

アセンブリ言語のプログラミングでは次の2点を絶対に忘れないでください。

- 疑似命令はマシンコードを生成しない。
- マシンコードはプログラマがソースコードで指示したとおりの処理しか行なわない。

RET命令を付け忘れる

サブルーチンの正常な呼び出しの過程は，サブルーチンコール，サブルーチンの処理実行，サブルーチンからのリターンの3ステップから成ります。サブルーチンを終了して呼び出し側プログラムに戻るためには，サブルーチンの最後に必らず**RET**命令を入れておかなくてはなりません。プログラムを入力する際は，次のように**RET**命令を忘れやすいので注意してください。

```
; 値を80倍するサブルーチン
; 入力: AX = 80倍される値
; 出力: DX:AX = 結果
;
MultiplyBy80    PROC NEAR
      mov   dx,80
      mul   dx
MultiplyBy80    ENDP

; 次のキー押下を取得するサブルーチン
; 入力: AL = 押されたキー
; AHは破壊される
;
GetKey      PROC NEAR
      mov   ah,1
      int   21h
      ret
GetKey      ENDP
```

この例では，サブルーチン*MultiplyBy80*の最後にENDP疑似命令があるために，*MultiplyBy80*が正常に終了すると勘違いしがちです。しかし実際に*MultiplyBy80*を呼び出すと，AXレジスタの内容を80倍するだけでなく，その下にある*GetKey*の処理も実行して(*AL*レジスタにキーコードを入れて)しまいます。*MultiplyBy80*が正しくリターンするためには次のようにしなければなりません。

```
; 値を80倍するサブルーチン
; 入力: AX = 80倍される値
; 出力: DX:AX = 結果
;
MultiplyBy80    PROC NEAR
     mov  dx,80
     mul  dx
     ret
MultiplyBy80    ENDP
```

リターンの型が間違っている

PROC疑似命令には2つ働きがあります。プロシージャの名前(プロシージャはこの名前で呼び出される）を定義することとそのプロシージャの型（nearまたはfar）を指定することです。

アセンブラはプロシージャの型（nearまたはfar）を参照して，それが同じソースファイルから呼び出される際にどの型のコールを実行するかを決めます。プロシージャ型は，プロシージャが呼び出し側へコントロールを戻すための**RET**命令の型を決める際にも使われます。次に例を示します。

```
; DX:AXを右へ2ビットシフトするためのNEARサブルーチン
;
LongShiftRight2   PROC NEAR
     shr  dx,1
     rcr  ax,1                    ; DX:AXを右へ1ビットシフト
     shr  dx,1
     rcr  ax,1                    ; 再びDX:AXを右へ1ビットシフト
     ret
LongShiftRight2   ENDP
```

この例では*LongShiftRight2*がnear型プロシージャであるため，**RET**命令はnear型になっています。**PROC**疑似命令を次のように変更すれば，RET命令はfar型となります。

```
LongShiftRight2   PROC FAR
```

ここまでの説明は理解できるはずです。ではプロシージャ内の**RET**命令の型は，プロシ

ージャの型と必らず一致しなければならないのでしょうか？

この答えは一概には決められません。問題となるのは同一のプロシージャ内の複数のサブルーチンを必要に応じてグループ化した場合です。このようなサブルーチンには対応するPROC疑似命令がないため，RET命令の型はそのプロシージャの型と同じになります(しかしその型は各サブルーチンに対して常に適正であるとは限りません)。次に例を示します。

```
; DX:AXを右へ2ビットシフトするためのNEARサブルーチン
;
LongShiftRight2    PROC FAR
     call LongShiftRight      ; DX:AXを右へ1ビットシフト
     call LongShiftRight      ; 再びDX:AXを右へ1ビットシフト
     ret
LongShiftRight:
     shr  dx,1
     rcr  ax,1                ; DX:AXを右へ1ビットシフト
     ret
LongShiftRight2    ENDP
```

この例は正常に動作しません。*LongShiftRight2*プロシージャがサブルーチン*LongShiftRight*を呼び出す際はnearコールを使っています。これは両者が同一のコードセグメント内にあるためです。しかしサブルーチン*LongShiftRight*は*LongShiftRight2*プロシージャ内にうめ込まれているため，サブルーチンの最後のRET命令はfarリターンになります。このようにnearコールとfarリターンを組み合わせて使うと，システムクラッシュが発生します。

この問題を解決するには，それぞれのサブルーチンに適した**PROC**疑似命令を使うことです。**PROC**疑似命令は次のようにネストすることもできます。

```
; DX:AXを右へ2ビットシフトするためのNEARサブルーチン
;
LongShiftRight2    PROC FAR
     call LongShiftRight              ; DX:AXを右へ1ビットシフト
     call LongShiftRight              ; 再びDX:AXを右へ1ビットシフト
     ret
LongShiftRight    PROC NEAR
     shr  dx,1
     rcr  ax,1                        ; DX:AXを右へ1ビットシフト
     ret
LongShiftRight     ENDP
LongShiftRight2    ENDP
```

次の例は上の**PROC**疑似命令を並列に並べたものです。

```
; DX:AXを右へ2ビットシフトするためのNEARサブルーチン
;
LongShiftRight2   PROC FAR
     call LongShiftRight             ; DX:AXを右へ1ビットシフト
     call LongShiftRight             ; 再びDX:AXを右へ1ビットシフト
     ret
LongShiftRight2   ENDP
LongShiftRight    PROC NEAR
     shr  dx,1
     rcr  ax,1                       ; DX:AXを右へ1ビットシフト
     ret
LongShiftRight    ENDP
```

RETNまたは**RETF**を使って，それぞれnearまたはfarリターンを行なうこともできます。これらの命令は**PROC**疑似命令で定義されたプロシージャの外でも使うことができます。これでも，なお常に正常にリターンすることができます。

オペランドの順番が逆になっている

多くの人にとって8086アセンブリ言語の命令のオペランドの順番は逆に見えます（この見方にも確かに一理あります)。たとえば

```
mov  ax,bx
```

という命令が"move AX to BX"（AXからBXへ動かす）を意味するのであれば，この命令は表記どおりに左から右に読むことができます。そして多くのマイクロプロセッサメーカは自社のアセンブリ言語の設計にあたって，この順番を採用しています。しかしインテル社は8086アセンブリ言語に逆の順番を採用しました。したがって上の命令は"move BX to AX"（BXからAXへ動かす）を意味します。これはしばしば混乱の元となります。

インテル社のオペランドの順番の背景にあるものは，CやPascalの場合の順番(つまりデスティネーションが左）と同じにするという考えです。8086アセンブリ言語のオペランドの1つの見方は，頭の中でオペランド間のカンマ（,）を等号（=）に置き換えることです。こうすると命令を代入式として見ることができます。たとえば

```
mov  ax,bx
```

という命令は

```
ax = bx
```

と考えるのです。そうすれば

```
add  bx,(OFFSET BaseTable * 4) + 2
```

のような定数オペランドを持つ命令は

```
bx += (OFFSET BaseTable * 4) + 2
```

と考えることができます。

スタックの確保を忘れる，またはスタック領域が足りない

スタック領域を明示的に確保しないでプログラムを走らせるのは，薄氷の上を歩くのと同じくらい危険なことです。スタック領域が確保されていないプログラムでも正常に動作することもあります。なぜならば，指定の無い場合はスタック領域としてメモリの未使用領域が使われるからです。しかしこのようなプログラムは，場合によってはスタック領域をまったく確保できない可能性もあり，あらゆる条件のもとで正しく動作する保証はどこにもありません。ほとんどのプログラムでは，.STACK疑似命令を使ってスタック領域を確保しておかなければなりません。この際，プログラムが使うと予想されるスタックサイズのよりもさらに大きな領域をスタック領域として確保しておく必要があります。

なぜこのように十分な領域を確保する必要があるのでしょうか。それはプログラムが使うスタック領域のサイズを予想するのが難しいためです。スタック領域が他のプログラムを侵して上書きしたときに発生するバグは，多くの場合再現・追跡するのが困難です。またデバッガの多くはプログラムからコントロールを得る際にスタック上の特別な領域を使います。したがってスタック領域を確保する際は，万一に備えて余裕のある大きめのサイズにしておくのがよいでしょう。スタックサイズの最小値の目安は512バイトです。

プログラムによってはスタック領域を確保する必要のないものもあります。それは.COMまたは.BINファイルになるプログラムです。.BINファイルには特定のアドレスから実行するためのマシンコードが入っています。また.BINファイルは通常BASICサブルーチンとして使われるため，BASICのスタックを使います。.COMプログラムは実行時にプログラムセグメントの先頭のスタックを使います（このセグメントの最大サイズは64Kバイトです）。したがってスタック領域の最大サイズは，プログラムセグメント内の未使用メモリのサイズとなります。スタック領域は伸び縮みするため，.COMプログラムのどれかのサイズが64Kバイトに近くならないか注意してください。またサイズの大きな.COMプログラムをメモリ容量の小さいコンピュータで走らせた場合，または他のプログラムからDOSシェルを使って走らせた場合は，スタックの問題が発生することがあります。これらの問題を避けるには，.COMプログラムではなく.EXEプログラムを作って，十分に余裕のあるスタック領域

を確保するのがもっとも確実な方法です。

サブルーチンコールによって必要なレジスタを破壊する

アセンブラコードを書いているときは，レジスタは現在のプロシージャだけが使用できるローカル変数であると思い込みがちです。特に他のプロシージャを呼び出してもレジスタの内容は変化しないものと思ってしまいます。しかしレジスタはグローバル変数です。したがってプロシージャによっては，一部またはすべてのレジスタを破壊することがあります。

次に例を示します。

```
        .
        .
        .
        mov   bx,[TableBase]       ; BXにテーブルベースをセット
        mov   ax,[Element]         ; 要素番号をロード
        call  DivideBy10           ; 10で割る
        add   bx,ax                ; 適切な項目をポイント
        .
        .
        .
;
; 値を10で割るサブルーチン
; 入力: AX = 10で割られる値
; 出力: AX = 10で割った結果
;       DX =余り
;
DivideBy10    PROC NEAR
        mov   dx,0                 ; DX:AXを32ビットの分子とする
        mov   bx,10                ; BXを16ビットの分母とする
        div   bx
        ret
DivideBy10    ENDP
```

この例では*DivideBy10*がBXに10をセットしているため，呼び出し側はこのサブルーチンによってBXレジスタは破壊されないと認識します。このような場合でも解決方法はいくつもあります。1つの方法として*DivideBy10*の入口でBXをプッシュ（退避）して出口でポップ（復帰）することです。次にその例を示します。

```
      .
      .
      .
      mov   bx,[TableBase]       ; BXにテーブルベースをセット
      mov   ax,[Element]         ; 要素番号をロード
      call DivideBy10            ; 10で割る
      add   bx,ax                ; 適切な項目をポイント
      .
      .
      .
; 値を10で割るサブルーチン
; 入力: AX = 10で割られる値
; 出力: AX = 10で割った結果
;       DX = 余り
; BXレジスタは破壊される
;
DivideBy10    PROC NEAR
      push bx                    ; BXを退避
      mov   dx,0                 ; DX:AXを32ビットの分子とする
      mov   bx,10                ; BXを16ビットの分母とする
      div   bx
      pop   bx                   ; BXを復帰
      ret
DivideBy10    ENDP
```

プッシュとポップは，呼び出し側プログラム内の*DivideBy10*のコールの前後でそれぞれ行なってもよいでしょう。次にその例を示します。

```
      .
      .
      .
      mov   bx,[TableBase]       ; BXにテーブルベースをセット
      mov   ax,[Element]         ; 要素番号をロード
      push bx                    ; BXを退避
      call DivideBy10            ; 10で割る
      pop   bx                   ; BXを復帰
      add   bx,ax                ; 適切な項目をポイント
      .
      .
      .
; 値を10で割るサブルーチン
; 入力: AX = 10で割られる値
; 出力: AX = 10で割った結果
;       DX =余り
; BXレジスタは破壊される
;
DivideBy10    PROC NEAR
      mov   dx,0                 ; DX:AXを32ビットの分子とする
      mov   bx,10                ; BXを16ビットの分母とする
      div   bx
      ret
DivideBy10    ENDP
```

BXへの値のセットをコールの前でなく，コールの後で行なう方法もあります。次にその例を示します。

```
        .
        .
        .
        mov   ax,[Element]        ; 要素番号をロード
        call  DivideBy10          ; 10で割る
        mov   bx,[TableBase]      ; BXにテーブルベースをセット
        add   bx,ax               ; 適切な項目をポイント
        .
        .
        .
; 値を10で割るサブルーチン
; 入力: AX = 10で割られる値
; 出力: AX = 10で割った結果
;       DX =余り
; BXレジスタは破壊される
;
DivideBy10   PROC NEAR
        mov   dx,0                ; DX:AXを32ビットの分子とする
        mov   bx,10               ; BXを16ビットの分母とする
        div   bx
        ret
DivideBy10   ENDP
```

レジスタを誤って破壊する問題はサブルーチンにはつきものです。この問題を防ぐもっとも簡単な方法は，常に無条件にすべてのレジスタをスタックに退避することです。ただしレジスタの退避・復帰を行なえば，その分だけ時間とメモリを消費するため，アセンブラプログラミングのメリットの一部を犠牲にすることになります。他の方法としてはサブルーチンの先頭に退避されるレジスタと破壊されるレジスタをコメントで示しておくことです。そしてサブルーチンをコールしたら，レジスタが退避されていることを常に注意深くチェックすることです。ただしこの方法では，呼び出し側プログラムで必要なレジスタを退避してやる必要があります。

条件ジャンプ命令を誤って使う

アセンブリ言語には多数の条件ジャンプ命令（**JE, JNE, JC, JNC, JA, JB, JG**など）が用意されており，コードを書く上で非常に柔軟性があります。そのかわりジャンプ命令を誤って使う可能性も高くなります。またアセンブリ言語による条件処理には，少なくとも2行が必要です。1行は比較のため，もう1行は条件ジャンプ命令です（複雑な条件処理ではより多くの行が必要になります）。そのためアセンブリ言語における条件処理は，CやPascalのそれに比べて直感的にわかりにくく，間違えやすいので，注意する必要があります。次によく見かける条件ジャンプ命令の誤った使用例を示します。

■JA, JB, JAE, JBEを符号付き数値の比較に使う。またJG, JL, JGE, JLEを符号なしの数値の比較に使う。

■JAEを使うべきときにJAを使う。JAE, JBE, JLE, JGEのように最後にEを付けないと，2つのオペランドが等しい場合を見落してしまいます。

■JNSを使うべきときにJSを使う。論理の反転には気をつけるべきです。

条件ジャンプ命令の誤りを防ぐためには，テストと条件ジャンプに関してC言語のようなコメントを付けることです。次に例を示します。

```
        .
        .
        .
;
; if ( Length > MaxLength ) {
;
        mov   ax,[Length]
        cmp   ax,[MaxLength]
        jng   LengthIsLessThanMax
        .
        .
        .
        jmp   EndMaxLengthTest
;
; } else {
;
LengthIsLessThanMax:
        .
        .
        .
;
; }
;
EndMaxLengthTest:
        .
        .
        .
```

ストリング命令の落とし穴

ストリング命令は8086の命令の中でも際立って強力なものです。しかしこの便利さがときとして問題を発生する原因となることがあります。次にそれらについて説明します。

REP文字列のオーバランを忘れる

ストリング命令の働きには注意すべき点があります。それは，命令実行後には使用ポインタは最後にアクセスしたアドレスより1バイト（ワード命令の場合は2バイト）先のアドレスを指していることです。次にこの例を示します。

```
.
.
.
cld                      ; ディレクションフラグをクリア(カウントアップ)
mov  si,0                ; オフセットに0をセット
lodsb                    ; オフセット0の値をロード
.
.
.
```

このプログラムを実行した後は，SIの内容は0ではなく1になります。これには理由があります。つまり次に**LODSB**命令を実行したときはアドレス1を，さらにその次に**LODSB**命令を実行したときはアドレス2にアクセスできるようにするためです。しかしストリング命令，特に**REP SCAS**や**REP CMPS**を繰り返し使用すると混乱を生じます。次にその例を示します。

```
.
.
.
cld                       ; ディレクションフラグをクリア(カウントアップ)
les  di,[bp+ScanString]   ; ES:DIに検索文字列アドレスをセット
mov  cx,MAX_STRING_LEN    ; 最長文字数をカウンタにセット
mov  al,0                 ; 終端のヌルコードをセット
repne scasb               ; 検索開始
.
.
.
```

このときESレジスタの内容が2000h, DIレジスタの内容が0，メモリの2000：0000番地からの内容が次のようになっているものと仮定します。

```
41h 61h 72h 64h 00h
```

この場合に上のプログラムを実行すると，DIレジスタの内容は5(検索されたNULL文字の次のアドレスのバイトオフセット）となります。ポインタが文字列の最後の文字を指すように戻すためには，次のようにプログラムを追加します。

```
      .
      .
      .
      cld                         ; ディレクションフラグをクリア(カウントアップ)
      les  di,[bp+ScanString]     ; ES:DIに検索文字列アドレスをセット
      mov  cx,MAX_STRING_LEN      ; 最長文字数をカウンタにセット
      mov  al,0                   ; 終端のヌルコードをセット
      repne scasb                 ; 検索開始
      jne  NoMatch                ; エラー終了(0見つからず)
      dec  di                     ; ポインタを0の位置へ戻す
      dec  di                     ; ポインタを最後の文字の位置へ戻す
      ret
NoMatch:
      mov  di,0                   ; ヌルポインタを返す
      mov  es,di
      ret
      .
      .
      .
```

ディレクションフラグをセットしたときはストリング命令によってDIはカウントダウンされ，DIレジスタは最後の文字の次ではなく，その前のアドレスを指します。

CXレジスタにも同様な混乱が生じます。つまり**REP SCAS**や**REP CMPS**を実行中に1回だけ余分にデクリメントされます。CXレジスタは「～である間は繰り返す」という条件（正論理でも負論理でも可）が成立するたびにデクリメントされますが，その条件が成立しないときもデクリメントされ，CXが0になった時点で命令の実行は終了します。

上の例で2000:0000番地の内容が0の場合，0以外の値がたとえ見つからなかった場合でも，実行終了後にはCXの内容は*MAX_STRING_LEN-1*となります。文字列内の文字数をカウントするサブルーチンはこのことを考慮して作る必要があります。次に例を示します。

```
; 0を終端とする文字列の長さをバイト数で返すサブルーチン
; 入力: ES:DI = 文字列の開始アドレス
; 出力: AX = 文字列の長さ(終了コード0含まず)
;       ES:DI = 文字列の最後のバイトを指す
;               終了コードがない場合は0000:0000を返す
;
StringLength   PROC NEAR
      cld                        ; ディレクションフラグをクリア(カウントアップ)
      push cx                    ; CXを退避
      mov  cx,0FFFFh             ; 最大文字数セット
      mov  al,0                  ; 検索の終了コード0
      repne scasb                ; 終了コード0をチェック
      jne  StringLengthError     ;終了コードがみつからなければエラー処理へ
      mov  ax,0FFFFh             ; 最大文字数セット
      sub  ax,cx                 ; カウントされたバイト数
      dec  ax                    ; 終了コードの分をカウントから除く
```

```
    dec  di              ; 終了コードにポインタ戻す
    dec  di              ; 最終文字にポインタ戻す
    jmp  short StringLengthEnd
StringLengthError:
    mov  di,0            ; 0000:0000をセット
    mov  es,di
StringLengthEnd:
    pop  cx              ; CXを復帰
    ret
StringLength   ENDP
```

CXをバイトカウンタとして使い，**REP SCAS**または**REP CMPS**を実行した場合は，たとえ終了条件が成立していたとしても，最後の比較処理でCXレジスタがすでに0になっているという別の問題が生じます。たとえば最終バイトだけが異なる2つの配列を比較した場合がそうです。この場合は，最終バイトの比較時にCXはカウントダウンしてすでに0になり，両配列が等しいかどうかを正しく評価できなくなります。次に例を示します。

```
.
.
.
repz  cmpsb
jcxz  ArrayAreTheSame
.
.
.
```

両配列の比較を正しく行なわせるためには，次のように直します。

```
.
.
.
repz  cmpsb
jz    ArrayAreTheSame
.
.
.
```

要するに，CXは**REP SCAS**や**REP CMPS**を使ったスキャンまたは比較処理に対してのみバイトカウンタとして使うことができます。CXはデータをスキャンまたは比較して2つのデータが等しいか否かをチェックするための指標としては使わないことです。

プログラムで繰り返しストリング命令を使った場合にその実行結果を予想しかねるときは，机上デバッグによって，またはデバッガを使って（1ステップずつ）繰り返しストリング命令の動作を追跡することをお勧めします。

CXカウンタを使えばセグメント全体をカバーできると思い込む

繰り返しストリング命令はCXが0のときは何もしません。このことは繰り返しストリング命令の実行前にCXが0かどうかチェックする必要がなくなるという意味では便利です。しかしバイトサイズのストリング命令でセグメント内の全バイトにアクセスすることができなくなります。次の例はESセグメントを先頭からスキャンして"*A*"という文字を見つけるプログラムです。

```
        .
        .
        .
        cld                        ; ディレクションフラグをクリア(カウントアップ)
        sub   di,di                ; オフセット0から開始
        mov   al,'A'               ; 'A'を検索
        mov   cx,0FFFFh            ; バイトカウンタをセット(64Kb)
        repne SCASb                ; 1バイトずつスキャン
        je    AFound               ; 'A'を発見
        scasb                      ; 見つからず...最終バイトをチェック
        je    AFound               ; 最終バイトに発見
        .                          ; 見つからず...このセグメント内に'A'はない
        .
        .
    AFound:                        ; DI = 'A'のアドレス+1
        .
        .
        .
```

8086の命令セットでは，CXカウンタの0の扱いに関して必らずしも統一がとれていません。繰り返しストリング命令はCXが0ならば何も行ないません。しかし**LOOP**命令はCXがたとえ0でも処理を行ない，CXをデクリメントして0FFFFhとし，ループアドレスへジャンプします。このことは1つのループで64Kb全体を処理できることを物語っています。上の例を**LOOP**命令を使って書き直すと次のようになります。

```
        .
        .
        .
        cld                        ; ディレクションフラグをクリア(カウントアップ)
        sub   di,di                ; オフセット0から開始
        mov   al,'A'               ; 'A'を検索
        mov   cx,0FFFFh            ; バイトカウンタをセット(64Kb)
    ASearchLoop:
        scasb                      ; 次のバイトをチェック
        je    AFound               ; 'A'を発見
        loop ASearchLoop           ; 見つからず...このセグメント内に'A'は無い
        .
        .
    AFound:                        ; DI = 'A'のアドレス+1
        .
        .
        .
```

CXが0のときは「何もしてはいけない」という重要な意味を持つ処理でLOOPを使う場合は，CXが0か否かをチェックすることは特に重要です。それ以外の場合に64Kバイト分のループ処理を使うと，悲惨な結果をもたらす可能性があります。**JCXZ**命令はそのような場合に役立ちます。

```
; 64Kbのメモリを指定したバイト値でパディングするサブルーチン
; 入力: AL = バイト値
;       CX = パディングするバイト数
;       DS:BX = パディング開始アドレス
; BX,CXは破壊される
;
FillBytes PROC NEAR
     jcxz FillBytesEnd        ; パディングバイト数=0なら終了
FillBytesLoop:
     mov  [bx],al             ; 1バイトストア
     inc  bx                  ; 次のアドレスをポイント
     loop FillBytesLoop       ; 指定バイト数だけ繰り返す
FillBytesEnd:
     ret
     FillBytes ENDP
```

この例では先頭に**JCXZ**が入れてあるために，CXが0のときはメモリの内容が変わらないで済みます。そうしないとCXが0の場合でもESで指定されたセグメント内の全アドレスにALの値がストア（パディング）されてしまいます。

ディレクションフラグの設定が間違っている

ストリング命令を実行したときは，ポインタSIとDIのどちらかまたは両方がインクリメントまたはデクリメントされます。インクリメントされるかデクリメントされるかはディレクションフラグの状態によって決ります。

ディレクションフラグは**CLD**命令にクリアされ，ストリング命令によってインクリメント（カウントアップ）されます。また**STD**命令にセットされ，ストリング命令によってデクリメント（カウントダウン）されます。ディレクションフラグはいったんクリアまたはセットされると，次に**CLD**または**STD**が実行されるまで（または**POPF**または**IRET**によりスタックからフラグがポップされるまで）変化しません。ディレクションフラグを一度設定すると，一方向にデータを動かすストリング命令をいくつでも実行できることは大変便利です。その反面，過去に実行されたコードによってはストリング命令が予想に反した処理を行なったり，間欠性で発見が困難なバグの原因になったりします。

この原因は何でしょうか？　通常カウントアップの方がカウントダウンよりも簡単で動作が確実であるため，ほとんどのプログラムではディレクションフラグをクリアしています。しかしカウントダウンを使う方が便利な場合もあります。一般的にストリングを使う

前はディレクションフラグを常にクリアしても，他のプロシージャ（このプロシージャ内でディレクションフラグをセットする）を呼び出した後では忘れがちです。その結果，カウントアップの処理はうまくいくがフラグの後処理をしないプロシージャの後では誤動作する，ということが起きます。

この問題の対処法は簡単です。ディレクションフラグは常に正しく設定されているとは限らないため，ストリング命令を使う前は常にディレクションフラグを所定の状態に設定することです。したがってストリング命令を使うプロシージャの先頭では，必らずディレクションフラグを設定してください(このルールは一般的プログラミング上での話で，Turbo C, Turbo Pascalなどでは通常，ディレクションフラグはクリアされており，ディレクションフラグをセットして使い終わったらクリアするようにします)。

繰り返しストリング比較命令を誤って使う

CMPS命令は2つのメモリアドレスの内容を比較します。一方**SCAS**命令はアキュムレータとメモリアドレスの内容を比較します。いずれの命令も前に**REPE**プレフィクスを付けると，CXレジスタが0になるか不一致が成立するまで比較処理を繰り返します。また**REPNE**プレフィクスを付けたときは，CXレジスタが0になるか一致が成立するまで比較処理を繰り返します。**REP**プレフィクスの意味は混乱しやすいので注意が必要です。

REPプレフィクスの機能を憶えておくいい方法は，頭の中で"rep"の後に"while"を入れることです。たとえば**REPE**なら"rep while e"つまり"repeat while equal"となります。同様に**REPNE**なら"rep while ne"つまり"repeat while not equal"となります。

ストリング命令のデフォルトセグメントを忘れる

ストリング命令を使う場合は，必要に応じてソースオペランド（転送元オペランド）とデスティネーションオペランド（転送先オペランド）を指定しなければなりません。ストリング命令では，デフォルトとして，DSレジスタのセグメントがソースで，ESレジスタのセグメントがデスティネーションとなっています。ストリング命令以外の命令では通常データセグメント内のデータを扱うために，このことを忘れてたとえば**STOSB**をデータセグメントに対して実行することがあります。次に誤ったプログラムの例を示します。

```
        .
        .
        cld                 ; ディレクションフラグをクリア(カウントアップ)
        mov   al,0
        mov   cx,80         ; バッファサイズ
        repe  scasb         ; 0以外の文字を検索
        jz    AllZero       ; 0以外の文字見つからず
        dec   di            ; 最初の0以外の文字にポインタを戻す
        mov   al,[di]       ; 最初の0以外の文字をロード
        .
        .
        .
AllZero:
        .
        .
        .
```

この例ではDSとESが同じ場合を除いて，最後の**MOV**命令は正しい値をALにロードしません。その理由は，**SCASB**命令はESが指すアドレスに対してデータをアクセスし，**MOV**命令はDSに対してデータをアクセスするからです。プログラムを正しく動作させるためには，セグメントオーバライドプレフィクスを使うことです（セグメントプレフィクスについては第10章を参照してください)。次の例はセグメントプレフィクスを使って書き直したプログラムです。

```
        .
        .
        .
        cld                 ; ディレクションフラグをクリア(カウントアップ)
        mov   al,0
        mov   cx,80         ; バッファサイズ
        repe  scasb         ; 0以外の文字を検索
        jz    AllZero       ; 0以外の文字見つからず
        dec   di            ; 最初の0以外の文字にポインタを戻す
        mov   al,es:[di] ; 最初の0以外の文字をロード
        .
        .
        .
AllZero:
        .
        .
        .
```

次のようにすれば，ストリング命令のソースセグメントのDSを一時的にESに変えることが可能です。

```
.
.
.
lods es:[SourceArray]
.
.
.
```

しかし，ストリング命令のデスティネーションセグメントのESを一時的に他のセグメントに変えることはできません。したがって次の命令は使えません。

```
.
.
.
stos ds:[DestArray]
.
.
.
```

(Turbo Assemblerでこれを行なうとエラーとなります)。

バイトサイズからワードサイズへの変換に誤りがある

通常はストリング命令に対してはできるだけ大きなデータサイズ(通常はワード，80386ではダブルワード）を使う方が有利です。通常はデータサイズが大きいほどストリング命令の処理速度は向上するからです。次に例を示します。

```
.
.
.
mov   cx,200       ; 転送バイト数
.
.
.
shr   cx,1         ; バイト数からワード数へ変換
rep   movsw        ; ブロックをワードサイズで転送
.
.
.
```

上のプログラムを8088で走らせると，次のプログラムより処理速度が約50%向上します。

```
        .
        .
        .
        mov   cx,200      ; 転送バイト数
        .
        .
        .
        rep   movsb       ; ブロックをバイトサイズで転送
        .
        .
        .
```

しかしいくつかの落とし穴もあります。たとえば次の命令

```
shr cx,1
```

によって単純にバイト数をワード数に変換しようとすると，CXが奇数のときは1バイトだけ小さくなります。この原因は最下位ビットが無視されるためです。CXが奇数の場合に備えて，次のように条件処理を行なうとよいでしょう。

```
        .
        .
        .
        shr   cx,1              ; ワードカウントに変換
        jnc   MoveWord          ; 奇数バイトカウントか?
        movsb                   ; Yes...奇数バイト転送
MoveWord:
        rep   movsw             ; ワードサイズ(偶数バイト)で転送
        .
        .
        .
```

SHRはシフト命令ですが，バイト値を2で割る命令でもあります。STOSWに対してワードカウントではなくバイトカウントを使うと他のデータを消してしまい，さまざまなトラブルの原因になります。次に例を示します。

```
        .
        .
        .
        mov   cx,200      ; 転送バイト数
        .
        .
        .
        rep   movsw       ; ブロックをワードサイズで転送
        .
        .
        .
```

この例では転送先ブロックの後の200バイト（100ワード）を消してしまいます。

複数のプレフィクスを使用する

ストリング命令に複数のプレフィクスを付けることは避けてください。エラーの原因となることがあります。次に例を示します。

```
.
.
.
rep  movs  es:[DestArray],ss:[SourceArray]
.
.
.
```

この例ではREPプレフィクスと，SSセグメントオーバライドプレフィクスの2つのプレフィクスを使っています。ストリング命令実行中にハードウェア割り込みが発生すると，複数のプレフィクスが付いていた場合は問題となります。インテル社のいくつかのプロセッサ（8086と8088を含む）では，割り込み処理終了後に複数のプレフィクスを使ったストリング命令に戻ったときに，最後のプレフィクス以外のものは無視されます。その結果，ストリング命令が所定の回数だけ実行されなかったり，誤ったセグメントがアクセスされることがあります。

ストリング命令でどうしても複数のプレフィクスを使わなければならない場合は，命令の実行中に割り込みが発生しないように割り込みをマスクしてください。次にその例を示します。

```
.
.
.
cli
rep  movs  es:[DestArray],ss:[SourceArray]
sti
.
.
.
```

ストリング命令のオペランドをあてにする

ストリング命令の（1つまたは複数の）オプションのオペランドは，データサイズ指定とセグメント指定（セグメントオーバライド）のためにのみ使われます。しかし実際に指示したメモリアドレスにアクセスされる保証はありません。次に例を示します。

```
      .
      .
DestArray dw  256 dup (?)
      .
      .
      .
      cld                       ; ディレクションフラグをクリア(カウントアップ)
      mov  al,'*'               ; パディングする値
      mov  cx,256               ; パディングするワード数
      mov  di,0                 ; パディング開始アドレス
      rep  stos es:[DestArray]  ; パディング実行
      .
      .
      .
```

この例ではアドレスの*DestArray*がどのセグメント内にあっても,ESセグメントのオフセット0から256バイトにアスタリスク(*)をセットします。*DestArray*はワード配列であるため,**ES:**［*DestArray*］はアセンブラに対して**STOSW**を使うことを指示するだけです。またストリング命令がアクセスするオフセットを決めるのは,オペランドではなくSIまたはDIです。ストリング命令にオプションのオペランドを使えば,バイト配列に対して誤ってワードサイズのアクセスを行なうこと防ぐのに役立ちます。

同様に**XLAT**（translate）命令のオプションのオペランドは,型のチェックとセグメントオーバライドのためにのみ使われます。次に例を示します。

```
      .
      .
      .
LookUpTable   LABEL BYTE
      .
      .
      .
ASCIITable    LABEL  BYTE
      .
      .
      .
      mov  bx,OFFSET ASCIITable         ; 参照テーブルアドレスをBXにセット
      mov  al,[CharacterToTranslate]    ; バイトオフセットをALにセット
      xlat [LookUpTable]                ; バイトをロード
      .
      .
      .
```

この例はアドレス*LookUpTable*ではなく,アドレス*ASCIITable*からALバイト目の内容をロードします。この命令のオペランドの役割は,処理がバイトサイズであることとセグメントオーバライドを指定することです。**XLAT**命令はそのオペランドにかかわらず,常にBX＋ALが示すオフセットの内容をロードします。

予期しない副作用が発生することもある

アセンブラプログラムは8086が直接理解することのできる言語です。そのためアセンブリ言語のプログラマにとって，プロセッサのレジスタやフラグの状態変化はもっとも興味のあることがらといえます。アセンブラプログラムでは，プロセッサの状態を直接変えることができます。次に例を示します。

```
add  bx,[Grade]
```

この例は，*Grade*というメモリアドレスの16ビット値とBXレジスタの値を加え，その加算の結果を示すためにオーバフロー，サイン，ゼロ，補助キャリ，パリティ，キャリの各フラグをセットします。命令によってはプロセッサの状態変化がほとんど見られないものもあります。次にそのような命令について説明します。

乗算によりレジスタを破壊する

乗算においては（8ビット×8ビット，16ビット×16ビット，32ビット×32ビットのどの場合でも）ソースオペランドとして使用されるアキュムレータ以外に，少なくとも1つのレジスタの内容が破壊されます。この現象は8ビットどうしの乗算結果が16ビットを，16ビットどうしの乗算結果が32ビットを，32ビットどうしの乗算結果が64ビットをそれぞれ超えた場合も必ず起こります。乗算のソースとデスティネーションを表5.1に示します。

表5.1 MUL命令とIMUL命令のソースとデスティネーション

ソースオペランドのビットサイズ	ソース 明示的オペランド	ソース デフォルトオペランド	デスティネーション 上位	デスティネーション 下位	使用例
8×8	*reg8**	AL	AH	AL	mul dl
16×16	*reg16***	AX	DX	AX	imul bx
32×32†	*reg32*‡	EAX	EDX	EAX,	mul esi

* *reg8*はAH, AL, BH, BL, CH, CL, DH, DLのどれか。

** *reg16*はAX, BX, CX, DX, SI, DI, BP, SPのどれか。

† 32×32の乗算は8086, 8088, 80186, 80188, 80286ではサポートされていない。

‡ *reg32*はEAX, EBX, ECX, EDX, ESI, EDI, EBP, ESPのどれか。

上の表で明示的に指定されているのは,2つのオペランドの一方と演算のビットサイズだけです。しかしMUL命令とIMUL命令を使う際はこれ以外にも注意しなければならない点があります。ソースオペランドとして使われているアキュムレータとデスティネーションとして使われているレジスタは，デフォルトの指定に過ぎません。この表からは，命令実行後に他のレジスタも破壊されることは見落されてしまいます。たとえば16×16ビットの乗算結果をストアするためには,AXレジスタだけ使ったのではオーバフローしてしまいます。このような場合はDXレジスタにも結果の一部がストアされるためDXレジスタも破壊されます。**MUL**または**IMUL**命令を実行すると,AL, AX, EAXのみではなくAH, DX, EDXも必らず破壊されることを憶えておいてください。

ストリング命令がいくつかのレジスタを書きかえることもある

ストリング命令（**MOVS, STOS, LODS, CMPS, SCAS**）のどれかを実行すると，特定のフラグと最大3つのレジスタに影響を及ぼします。**MUL**命令と同様，ストリング命令による多くの影響はそのオペランドに明示されません。ストリング命令を使うときは，SIとDIの一方または両方が（ディレクションフラグの状態によって）インクリメントまたはデクリメントされることを忘れないでください。ストリング命令に**REP**プレフィクスを付けて実行した場合は,CXが0になるまで最低1回はデクリメントされることに注意してください。

キャリフラグを変化させない命令もある

予想に反してレジスタやフラグに影響を及ぼす命令がある一方で，予想に反してどのフラグにも影響を及ぼさない命令があります。次に例を示します。

```
inc  ah
```

この命令は次の命令と論理的には同じように思われます。

```
add  ah,1
```

両者の間には1つだけ違いがあります。**ADD**命令は演算結果が大きすぎる場合はキャリフラグをセットしますが，**INC**命令ではそういうことはありません。したがって次のプログラムは，DX:AXにストアされている32ビットの値をインクリメントしたことになります。

```
    .
    .
    .
    add  ax,1
    adc  dx,0
    .
    .
    .
```

しかし次のプログラムはそうなりません。

```
    .
    .
    .
    inc  ax
    adc  dx,0
    .
    .
    .
```

DEC命令もフラグに影響を与えますが, **LOOP, LOOPZ, LOOPNZ**などはフラグに影響を与えません。状況によっては，これらの命令は現在のフラグの設定を変えないで実行できるためたいへん便利です。重要なことはそれぞれの命令の動作を正確に把握することです。命令がフラグにどんな影響を与えるのか予想できないときは，必らず事前に調べましょう。

フラグの設定場所と参照場所が離れすぎている

フラグの状態は，フラグに影響を与える命令が次に実行されるまで変化しません。通常はフラグをセットする命令とフラグを参照する命令は比較的近くに置きます。そうすればフラグがセットされた場所からできるだけ近いところでそのフラグを参照することになり，バグ発生の確率を最小に抑えることができます。よくあるフラグの使用例として，条件チェック，1つまたは複数レジスタの設定，フラグの状態によるジャンプ，という処理パターンがあります。次に例を示します。

```
    .
    .
    .
    cmp  ax,1
    mov  ax,0
    jg   HandlePositive
    .
    .
    .
```

この例はAXレジスタの状態をチェックし，それを強制的に0にセットしてから所定のス

データス処理ルーチンへジャンプしています。では次に別の例を見てください。

```
  .
  .
  .
  cmp  ax,1
  sub  ax,ax
  jg   HandlePositive
  .
  .
  .
```

この例は前の例より簡潔で速度が速く，一見優れているように思われます。しかし比較命令（CMP）によってセットされたフラグの状態は，次の減算命令ですべて変わってしまいます。フラグのセット場所と参照場所が離れたために不都合が起る典型的な例です。

メモリオペランドと即値オペランドを混同する

アセンブリ言語では，メモリ変数またはメモリ変数にストアされた値をオフセット値として参照します。しかしメモリ変数の参照と値の参照の区別に関しては若干わかりにくいところがあるため，メモリ変数のオフセット参照と値参照はよく混同されます。

図5.1はメモリ変数のオフセットと値の違いを示します。ワードサイズの変数*MemLoc*のオフセットは5002hでその値は1234hです。

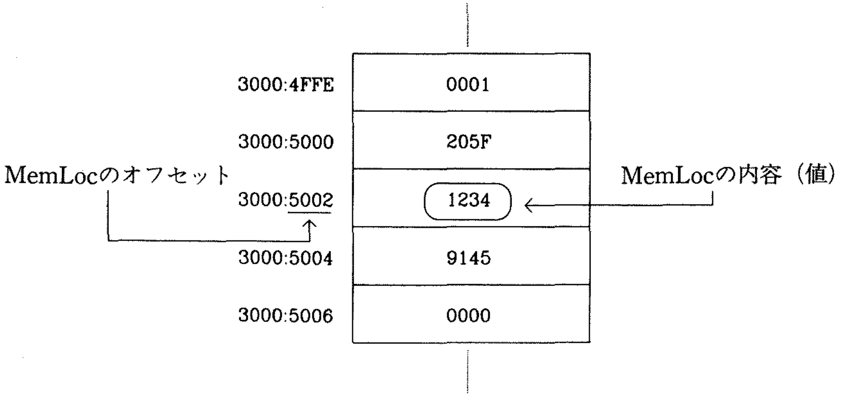


図5.1　メモリ変数のオフセットと値

図5.1ではワードサイズの変数*MemLoc*のオフセットは定数5002hで，**OFFSET**演算子によってロードすることができます。次の例はBXレジスタに5002hをロードします。定数5002h

は即値オペランド（つまり命令に組み込まれて変化しない定数）です。

```
mov   bx,OFFSET MemLoc
```

*MemLoc*の値は1234hです。この値はデータセグメント内のオフセット5002hのメモリの内容です。この値のロード方法はまずBX, SI, DI, BPレジスタのどれかに*MemLoc*のオフセットをストアし，このレジスタをアドレスポインタとすることです。次に例を示します。

```
mov   bx,OFFSET MemLoc
mov   ax,[bx]
```

この例は*MemLoc*の内容，1234hをAXレジスタにロードします。また次のどちらかの命令を使えば，AXに直接1234hをロードすることができます。

```
mov   ax,MemLoc
```

あるいは

```
mov   ax,[MemLoc]
```

ここで1234hは即値オペランドではなくダイレクトオペランドとしてロードされます。**MOV**命令はオフセット5002hが指定されており，AXに5002h番地の内容（この例では1234h）をロードします。もちろん1234hは他の値に変更できます。次に例を示します。

```
mov   [MemLoc],5555h
mov   ax,[MemLoc]
```

この例では1234hではなく5555hがAXにロードされます。

*MemLoc*のオフセットはデータセグメント内の特定のアドレスを表わす定数ですが，*MemLoc*の値はそのアドレスにストアされた変更可能な値です。次の例では*MemLoc*の内容は3となります。

```
mov   [MemLoc],1
add   [MemLoc],2
```

次の例は，

```
add   OFFSET MemLoc,2
```

次と同じですが，

```
add   5002h,2
```

ADD命令は2つの定数を加えることができないため意味を持ちません。プログラミングにおける共通の誤りとして，**OFFSET**を付け忘れることがあります。次に誤りの例を示します。

```
mov   si,MemLoc
```

この例では，*MemLoc*の前に**OFFSET**演算子を付ける必要があるにもかかわらず付けていません。またこの使いかたは一見間違いではないように見えます。事実*MemLoc*はワードサイズの変数であり，この例はエラーとはなりません。しかしこの命令を実行すると，SIレジスタには*MemLoc*のオフセット(図5.1の5002h)はロードされず，期待に反して*MemLoc*の内容（図5.1の1234h）がロードされます。

この問題を解決する決定的な手段はありません。しかしメモリ参照は常に大カッコ([])で囲むように習慣づければ，上のような誤りを防ぐことができます。アドレス定数を参照する場合は**OFFSET**プレフィクスを付け，メモリの内容を参照する場合は大カッコで囲むのがよいでしょう。このようにすればメモリ変数名のあいまいな使い方を避けることができます。先ほどの例

```
mov   si,OFFSET MemLoc
```

を使う際は注意すべきです。上のような習慣に従えば，アドレス参照と値参照は次のように，

```
mov   si,[MemLoc]
```

また値参照は次のようになります。

```
mov   si,MemLoc
```

セグメントレジスタの一巡エラー

8086のプログラミングの不便なところは，メモリ全体をサイズの大きな1つのバイトの配列としてアクセスできないことです。しかしセグメントレジスタをベースとすれば，64Kbのメモリ単位（セグメント）で連続的にメモリにアクセスすることができます。プログラムがあるセグメントの終端を越えて，さらに続きのメモリアドレスにアクセスしようとすると，アドレスポインタは再びそのセグメントの先頭アドレスに戻ります（つまりセグメントレジスタは0に戻ります)。このような，セグメントに起因するバグは発見がきわめて困難です。

例としてメモリの10000h以降のアドレスの内容が図5.2のようになっている場合を考えてみましょう。DSレジスタの内容が1000hで，アドレス1000:FFF9から文字列"*Testing*"がストアされているものとします。この場合アドレス1000:FFFFの"*g*"をアクセスした後にさらに次のアドレスにアクセスしようとすると，セグメントの先頭アドレス1000:0000に戻ってアクセスします。この原因はオフセットの最大値は0FFFFh（16ビットで表わせる最大値）だからです。

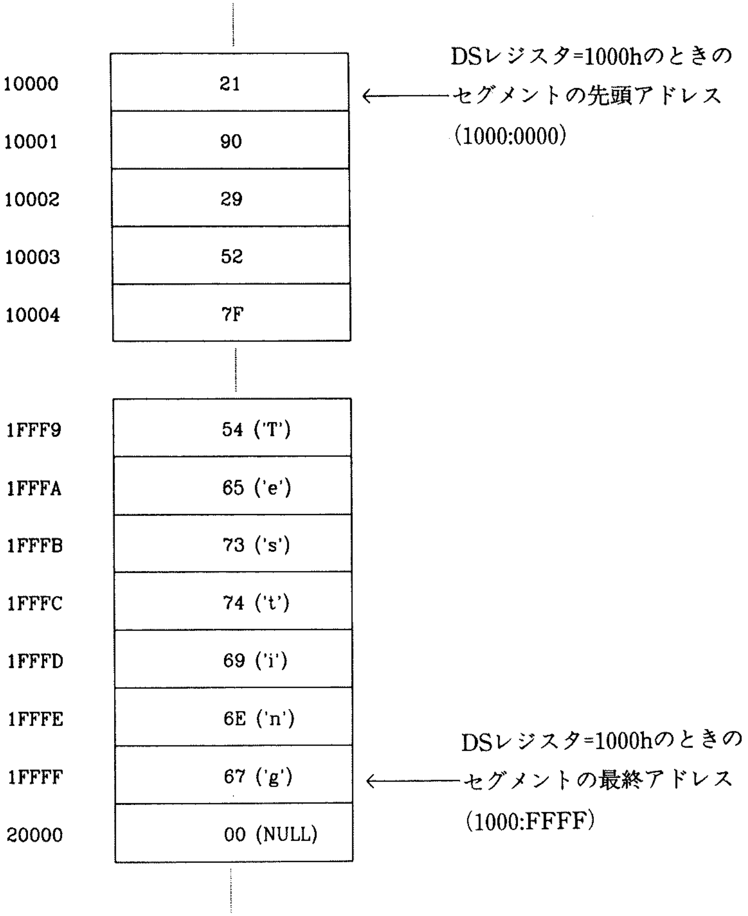


図5.2　セグメントレジスタの一巡エラーの例

では次に例を示します。下のサブルーチンは1000:FFF9からストアされている文字列"*Testing*"を大文字に変換するサブルーチンで，DS:SIが1000:FFF9のときに呼び出されます。

```
; 0で終了する文字列を大文字に変換するサブルーチン
; 入力: DS:SI =文字列の先頭アドレス
;
ToUpper   PROC NEAR
     mov  al,[si]            ; 次の文字をロード
     cmp  al,0               ; 0か?
     jz   ToUpperDone        ; Yes...文字列終わり
     cmp  al,'a'             ; 小文字か?
     jb   ToUpperNext        ; No...(小文字でない)
     cmp  al,'z'
     ja   ToUpperNext        ; 小文字でない
     and  al,NOT 20h         ; 小文字の場合は大文字に変換
     mov  [si],al            ; 大文字をストア
ToUpperNext:
     inc  si                 ; 次の文字をポイント
     jmp  ToUpper
ToUpperDone:
     ret
ToUpper   ENDP
```

*ToUpper*が目的の文字列の7文字すべてを処理した後, SIレジスタはインクリメントされ0FFFFhから0に一巡します (SIは16ビットレジスタのため最大0FFFFhしかカウントできません)。この際アドレス20000hにストアされているヌルコード (00) は文字列の終端を示すだけでアクセスされることはありません。したがって*ToUpper*は, 引き続きアドレス10000hのデータから順番に大文字への変換処理を実行し, ヌルコード (00) に出会うまでその処理を続行します。しかしデータ化けが起きているわけですから, いつかは必らずエラーが生じてプログラムはうまく動作しなくなります。このようなセグメントレジスタの一巡エラーによるデータ化けに起因するバグは, 概して追跡が困難です。この種のエラーは原因と症状の発生時間が大幅にずれており, ソースコード内の無関係な位置で発生することが多いためです。

この種のエラーの簡単な予防法はありません。プログラムが予想に反してセグメントの終端を超えてメモリにアクセスしないように注意するしかありません。またオフセット0FFFFhのワードをアクセスすることは非常に危険です(つまり, FFFFh番地と0000h番地をワードデータとして扱うこと)。もしそうするとコンピュータはハングアップします。

割り込みハンドラでレジスタやフラグの退避・復帰を行なわない

割り込みハンドラはキーボード割り込みなどのハードウェア割り込みが発生した場合に実行されるルーチンです。割り込みハンドラは押下キーのバッファリング, システムクロックの更新などのさまざまな処理を行ないます。割り込みはプログラム実行中にいつでも発生するため, 割り込みハンドラの先頭ではすべてのレジスタとフラグを退避し, 処理終

了時にそれらを元どおりに復帰する必要があります。レジスタとフラグの退避・復帰処理を怠ると，プロセッサ実行中に割り込みが発生すると突然プロセッサの状態が変っわてしまいます。次に例を示します。

```
.
.
.
mov  ax,[ReturnValue]
ret
.
.
.
```

プログラムを実行中に，この2つの命令の間で割り込みが発生することも考えられます。この場合に割り込みハンドラがAXレジスタの内容を退避していないと，呼び出し側プログラムにリターンした際にAXレジスタの値は割り込みハンドラの処理によって変化してしまいます。したがってAXレジスタにロードされていた*ReturnValue*変数の内容はまったく期待していない値になってしまいます。

したがって割り込みハンドラは，常にすべてのレジスタの内容を退避しなければなりません。ハンドラ内で破壊されるレジスタのみを退避すればそれで十分なのですが，ハンドラの先頭ですべてのレジスタをプッシュして，その出口でポップする方が確実です。なぜなら必要に応じて割り込みハンドラを変更した場合は，破壊されるレジスタが新たに増えているにもかかわらず，それらのレジスタを退避するのを忘れる可能性があるからです。割り込みハンドラ内でフラグを退避する必要はありません。割り込みが発生すると，フラグは自動的にスタックにプッシュされるからです。また割り込みハンドラが**IRET**命令を実行して元のプログラムに戻ると，フラグは自動的にスタックから復帰されます。

すべてのレジスタを退避する必要があるのは，割り込みハンドラの処理に入るとレジスタやフラグの状態がどうなるか予想できない場合です。この典型的な例は，**割り込みハンドラでディレクションフラグを設定しないでストリング命令を実行する場合**です。割り込みはプログラム実行中のどの時点で発生するかわかりません。したがってレジスタを退避したら，何よりも先に（セグメントレジスタを含む）レジスタとフラグの初期設定を行なわなければなりません。

オペランドおよびデータテーブルにグループオーバライドを付け忘れる

セグメントのグループ化の概念は理解しやすく，便利なものです。複数のセグメントを1つのグループにしたり，リンカを使ってこれらのセグメントを1つのセグメントに結合す

ることができます。この際グループ化されたすべてのセグメント内のすべてのデータは，特定のセグメントレジスタに対する相対アドレス（オフセット）としてアクセスできます。図5.3は3つのセグメント，*Seg1*, *Seg2*, *Seg3*をグループ化して*GroupSeg*というセグメントにしたようすを表わします。これらのセグメントは，*GroupSeg*のベースアドレスがストアされている特定のセグメントレジスタに対する相対アドレス（オフセット）として同時にアクセスすることができます。

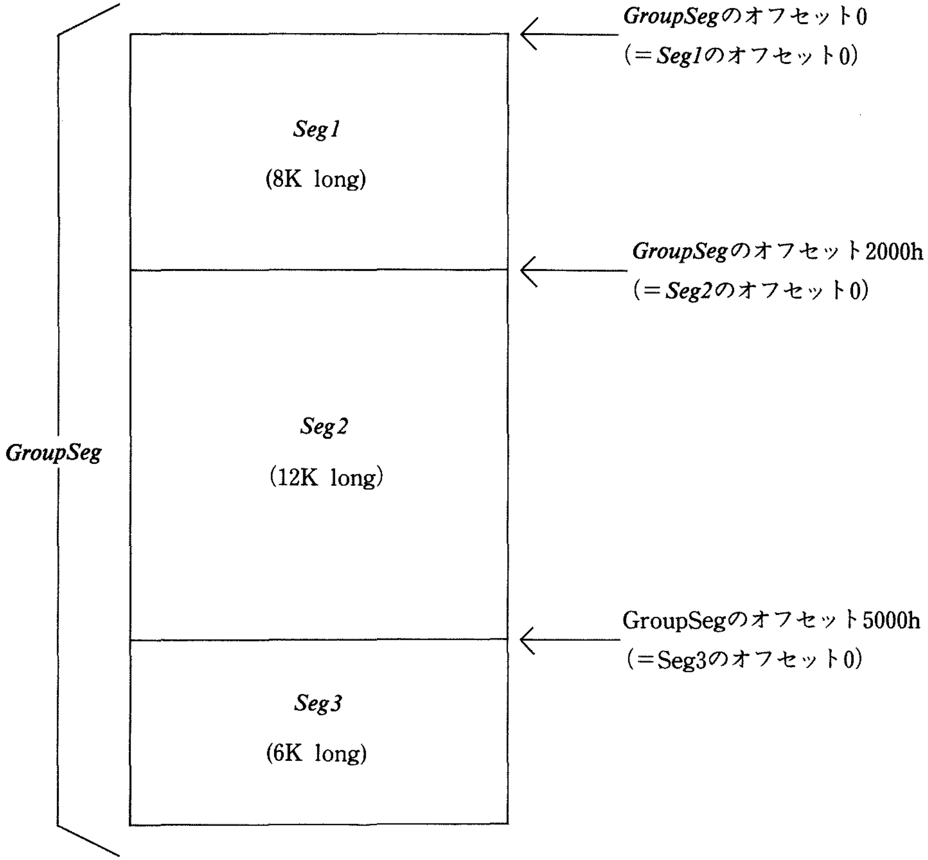


図5.3　3つのセグメントのグループ化

セグメントのグループ化によって，データを複数の領域に論理分割することができます。こうすれば，ある領域から他の領域に切り換えるたびにセグメントレジスタをロードする必要がなくなります。

マイクロソフト社のマクロアセンブラ（MASM）におけるセグメントグループの扱いか

たには若干の問題があります。したがってTurbo Assemblerが世に出るまでは，アセンブラでセグメントのグループ化を行なうのは困難でした。しかしアセンブラプログラムをC言語などの高級言語とリンクする際は，セグメントのグループ化が避けられない問題となります。

Turbo AssemblerのIdealモードを使えば，このセクションで説明するような問題が起ることはありません。MASM方式からこのIdealモードへ切り換えるべき理由はこの他にもあります。

MASMのセグメントグループに関する問題とは次のようなものです。つまり，**OFFSET**演算子によって得られるグループ化されたセグメント内のオフセットは，そのセグメントグループのオフセットではなく，各セグメント内のオフセットとして扱われます。たとえば図5.3に示すセグメントグループでは，*Var1*が*Seg2*の先頭に置かれていたとすると

```
mov   ax,OFFSET Var1
```

という命令は

```
mov   ax,0
```

としてアセンブルします。なぜならば*Var1*は*GroupSeg*内ではオフセット2000hですが，*Seg2*内ではオフセット0に相当するからです。セグメントグループ内のデータは，各セグメント内の相対アドレスではなくセグメントグループ内の相対アドレスを使ってアクセスされるため，これは問題となります。

この問題に対処するにはグループオーバライドプレフィクスを使うことです。次の命令はそれを使った例です。

```
mov   ax,OFFSET GroupSeg:Var1
```

この例では，*Var1*のオフセットはセグメントグループ*GroupSeg*内の相対アドレス(オフセット）として計算します。

この他にも，MASMにはセグメントグループで使われるデータテーブルに関して同様な問題があります。**OFFSET**演算子と同様，データテーブルとしてアセンブルされるオフセットは，セグメントグループ内ではなく各セグメント内の相対アドレスとなります。次に例を示します。

```
Stack    SEGMENT WORD STACK 'STACK'
     DB  512 DUP(?)                     ; スタック内に1/2Kb確保
Stack    ENDS
;
; Data1,Data2で構成されるデータセグメントグループDGROUPを定義
;
DGROUP   GROUP     Data1, Data2

;
; DGROUP内の第1セグメント
;
Data1       SEGMENT WORD PUBLIC 'DATA'
Scratch     DB  100h DUP(0)             ; 256バイトのスクラッチバッファ
Data1       ENDS

;
; DGROUP内の第2セグメント
;
Data2       SEGMENT WORD PUBLIC 'DATA'
Buffer      DB   100h DUP('@')          ; 256バイトのバッファ(@をセット)
BufferPtr   DW   Buffer                 ; バッファをポイント
Data2       ENDS

Code SEGMENT PARA PUBLIC 'CODE'
     ASSUME   CS:Code, DS:GFROUP

;
Start       PROC NEAR
        mov  ax,DGROUP
        mov  ds,ax                      ; DSにDGROUPをロード
        mov  bx,OFFSET DGROUP:Bufferptr ; バッファをポイント
                                        ; 注:DGROUP:グループオーバライドは
                                    ; 正しいオフセットを得るために必要
    mov  bx,[bx]                        ; バッファ自身をポイント
;
;
;
    mov  ah,4Ch                         ; プログラム終了のDOSファンクション
    int  21h                            ; プログラム終了, DOSへ戻る
Start    ENDP
Code ENDS
     END  Start
```

この例では

```
  mov  bx,OFFSET DGROUP:BufferPtr
```

内の*BufferPtr*のオフセットは，**DGROUP:**グループオーバライドプレフィクスが使われているため正しくアセンブルされます。次に別の例を示します。

```
BufferPtr DW    Buffer
```

この例は*BufferPtr*の値を*Buffer*のオフセットに初期化すべき例で，正しくアセンブルされません。なぜならば*Buffer*のオフセットは，セグメントグループ**DGROUP**内のオフセットではなく，セグメント*Data2*内のオフセットとなるためです。この問題も**DGROUP**オーバライドプレフィクスを使って次のように書き換えれば解決します。

```
BufferPtr DW    DGROUP:Buffer    ; バッファをポイント
                                 ; 注意: DGROUP:グループオーバライドは
                                 ;       正しいオフセットを得るために必要
```

MASMのQuirksモードでセグメントグループを使った場合にグループオーバライドプレフィクスを省略すると，プログラムがメモリの無関係な場所にアクセスしたり，予期せぬところへジャンプしたりして致命的なエラーが発生する可能性があります。基本ルールとして，やむをえない場合を除いてMASMではQuirksモードでセグメントグループを使わないことです。MASMのQuirksモードでセグメントグループを使わなければならない場合(高級言語とアセンブラプログラムを組み合わせたい場合など）は，グループ化されたセグメント内のオフセット指定時には，常にグループオーバライドプレフィクスを付けてください。グループオーバライドはたいへん使いやすいものです。憶えておくとよいでしょう。

MASMのQuirksモードでセグメントグループを扱う場合に役に立つテクニックとして，**MOV OFFSET**のかわりに**LEA**命令を使うことがあります。次に例を示します。

```
lea   ax,Var1
```

この命令にはグループオーバライドプレフィクスがありませんが，

```
mov   ax,OFFSET GroupSeg:Var1
```

を使った場合と同じ効果があります。ただし**LEA**命令の方が**MOV OFFSET**よりも1バイトだけサイズが大きく，実行速度が若干落ちます。

ところでセグメントグループの問題はメモリの内容をアクセスするときではなく，オフセットにアクセスする場合にのみ発生します。したがって次の命令にはグループオーバライドプレフィクスを付ける必要はありません。

```
mov   ax,[Var1]
mov   ax,OFFSET GroupSeg:Var1
```

第6章

Turbo AssemblerとTurbo Cのインターフェース

プログラムの作り方はプログラマによってさまざまです。あるプログラマはプログラム全体をアセンブリ言語だけで書き，あるプログラマはプログラムのほとんどは高級言語で書いて，低レベル制御や高速処理が要求されるところだけをアセンブリ言語で書きます。また基本的にはアセンブリ言語を使い，必要に応じて高級言語のライブラリやモジュールを利用するプログラマもいます。

Turbo Cでは，必要に応じてCプログラム内にアセンブラプログラムを組み込む場合に特に威力を発揮します。アセンブラプログラムとCプログラムを結合するには2通りの方法があります。1つはインラインアセンブラを使う方法です。この方法では，手軽にかつ容易にCプログラム内に直接アセンブラプログラムを挿入することができます。プログラムを複数のモジュールに分割し，すべてのモジュールをアセンブリ言語で書きたい場合は，アセンブラ のモジュールを個別にアセンブルしてTurbo Cプログラムにリンクすることができます。

この章で説明する内容は，Turbo Cのインラインアセンブラの使用法，個別にアセンブルしたTurbo AssemblerモジュールのTurbo Cプログラムへのリンク方法，Turbo CプログラムからのTurbo Assembler関数の呼び出し方法，Turbo AssemblerプログラムからのTurbo C関数の呼び出し方法です（**注意**：このマニュアルでTurbo Cに言及した場合は，バージョン1.5以降のTurbo Cを表わします)。

Turbo Cでインラインアセンブラを使う

もしアセンブラ使ってCプログラムを改良したいのなら，Cプログラムの中にアセンブラ命令を挿入することをお勧めします。スピードが要求される場所や低レベル操作が必要な場所にアセンブラプログラムを組み込めば，プログラムの性能が飛躍的に向上することでしょう。しかしたとえそうしたいと思っても，従来のような複雑な方法でアセンブラとCを組み合わせて使うのはめんどうなことです。またできることならCプログラムには何の変更も加えずにこの作業を行ないたいものです。うまく動いているプログラムは，できることなら修正すべきではないでしょう。

Turbo Cのインラインアセンブラは，これらのすべての要求を満たしてくれます。インラインアセンブラとは，Cプログラムの任意の場所に任意のアセンブラ命令を挿入する機能に他なりません。Cの定数，変数，関数にアセンブラから自由にアクセスできることはいうまでもありません。インラインアセンブラは純粋アセンブラに匹敵するほどパワフルなため，プログラムの改良以外の目的にも使うことができます。Turbo Cのライブラリにある高性能モジュールは，インラインアセンブラで書かれています。インラインアセンブラはCプログラム内に好きな量だけ入れることができます。この際，両言語の結合の方法を心配する必要もありません。

次のプログラムはCプログラム内にインラインアセンブラの命令文を挿入した例です。

```
.
.
.
i = 0;                   /* iに0をセット (C)
asm  dec  WORD PTR i;    /* iをデクリメント (アセンブラ)
i++;                     /* iをインクリメント (C)
.
.
.
```

最初と最後の行はC言語の普通の命令ですが，その間の行は違います。もうおわかりと思いますが，**asm**で始まる行が**インラインアセンブラ**の命令文です。デバッガを使ってこのプログラムの実行可能プログラムを調べると，次のようにコンパイルされることがわかります。

```
.
.
.
mov  WORD PTR [bp-02],0000
dec  WORD PTR [bp-02]
inc  WORD PTR [bp-02]
.
.
.
```

このプログラム例では**DEC**命令が

```
i = 0;
```

と

```
i++;
```

の間にはさまれてコンパイルされています。

Turbo Cのコンパイラはソースプログラム中に**asm**キーワード（インラインアセンブラの命令文であることを表わします）を見つけると，基本的にはそのアセンブラ行を直接コンパイルコードに変換します。この際，上の例の*i*の参照を**WORD PTR**［**BP-02**］に変換したのと同様に，Cの変数の参照を適切なアセンブラの同等の命令に変換します。つまりasmキーワードを使えば，Cプログラム内の任意の場所にアセンブラの命令文を挿入することができます（インラインアセンブラの動作にはいくつかの制限があります。これらについては少し後の「インラインアセンブラの制限」の節で解説します）。

アセンブラコードを直接Turbo Cのコードに変換することには少し危険が伴います。事実インラインアセンブラの使用にはそれなりのリスクが伴います。Turbo Cがソースをコンパイルする際は，インラインアセンブラの使用に伴う多くの危険な動作を回避するようになっています。しかしインラインアセンブラを誤って使用すると，重大なバグの原因になることは否めません。

ところでインラインアセンブラであれ純粋なアセンブラモジュールであれ，できの悪いアセンブラプログラムは暴走の原因となります。これはアセンブリ言語の高速処理と低レベル操作のメリットの代償と考えるべきでしょう。純粋なアセンブラプログラムよりもインラインアセンブラのプログラムの方がバグの発生する確率は低いでしょう。なぜならばTurbo Cは関数呼び出し，関数からのリターン，パラメータの受け渡し，変数領域の確保などの多くのプログラミングの要因を肩がわりしてくれるからです。インラインアセンブラを使えばCプログラムを比較的簡単に改良またはパワーアップできます。たとえその代償としてアセンブラのバグに煩わされても十分みあうものです。

次にインラインアセンブラを使う際の注意点を示します。

1. インラインアセンブラを使う場合はTCC.EXE（Turbo Cのコマンドラインバージョン）を起動してください。TC.EXE（Turbo Cのユーザインターフェースバージョン）はインラインアセンブラをサポートしていません。
2. Turbo Assemblerに付属しているTLINKとTurbo Cに付属しているTLINKはバージョンが異なる可能性があります。TLINKにはTurbo Assemblerをサポートするための重要な改良が施されました。またこれ以上の改良は予定されていません。したがってインラインアセンブラコードを含むTurbo CモジュールをTLINKでリンクする際は，TLINKの最新バージョンを使ってください。またリンカ用のディスク上のTLINK.EXEファイルは必らず1つだけにしてください。TLINK.EXEはボーランド社の製品に付属しているもののうちの最新バージョンを使ってください。

インラインアセンブラの使いかた

通常Turbo CはC言語で書かれたソースファイルをコンパイルして直接オブジェクトファイルに変換します。次にTLINKを起動して，複数のオブジェクトファイルをリンクして実行可能プログラムを作ります。図6.1はこれらのコンパイルとリンクのプロセスを示します。このプロセスはコマンドラインからはじめることができます。次に例を示します。

```
tcc filename
```

この例はFILENAME.CをコンパイルしてFILENAME.OBJを作り，次にTLINKを起動してFILENAME.OBJをリンクしてFILENAME.EXEを作ることを指定します。

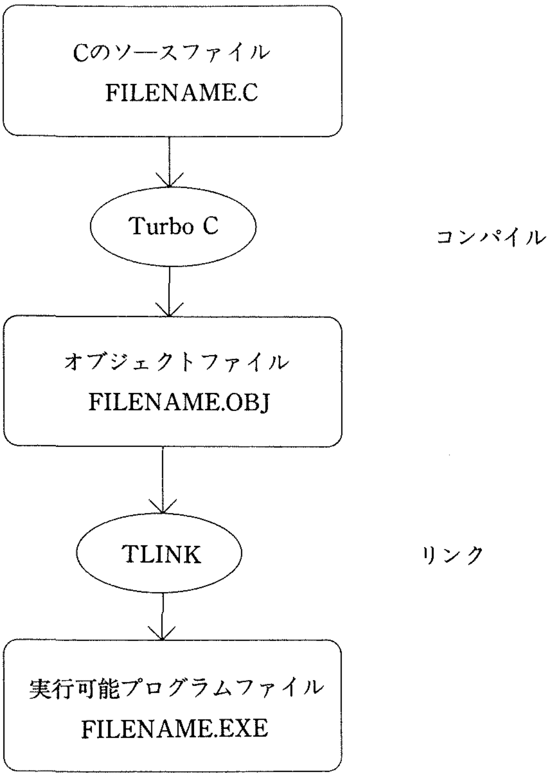


図6.1 Turbo Cのコンパイルとリンクのプロセス

ただしインラインアセンブラを使った場合は，このコンパイル-リンクプロセスにもう1ステップが加わります。

Turbo Cはインラインアセンブラの命令文を含んだファイルをまずアセンブリ言語のソースファイルにコンパイルし，次にTurbo Assemblerを呼び出してそのアセンブラファイルをオブジェクトファイルに変換します。最後にTLINKを呼び出してオブジェクトファイルをリンクします。図6.2にこのプロセス（Turbo Cがインラインアセンブラの命令文を含むCのソースファイルから実行可能プログラムを作るまでのプロセス）を示します。このプロセスを開始するには次のようなコマンドラインを使います。

```
tcc -b filename
```

この例はFILENAME.CをコンパイルしてFILENAME.ASMを作り，Turbo Assemblerを起動してFILENAME.ASMをアセンブルしてFILENAME.OBJを作り，最後にTLINKを起動してFILENAME.OBJをリンクしてFILENAME.EXEを作ることを指定します。

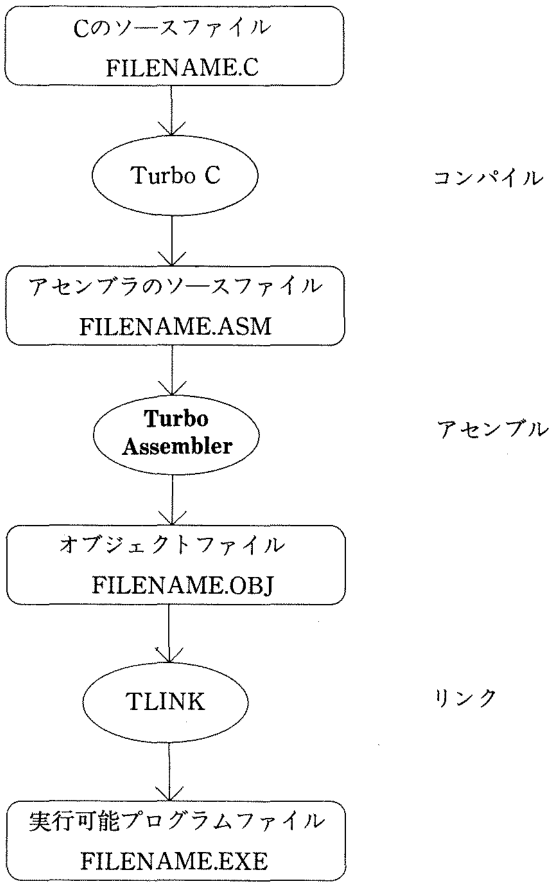


図6.2　Turbo Cのコンパイル,アセンブル,リンクのプロセス

インラインアセンブラコードはTurbo Cからアセンブリ言語のファイルとして渡されます。この方式のいい点は，Turbo Cがインラインアセンブラコードのアセンブルについて何1つ関与する必要がないことです。Turbo CはCコードをインラインアセンブラコードと同じレベル（アセンブラレベル）までコンパイルし，そのアセンブルをTurbo Assemblerに任せています。

Turbo Cでのインラインアセンブラの取り扱いを調べるために，次のプログラムを入力してファイルPLUSONE.Cにセーブしてください。

```
#include <stdio.h>

int  main(void)
{
    int  TestValue;

    scanf("%d",&TestValue);          /* インクリメントする値を取得*/
    asm  inc  WORD PTR TestValue;    /* インクリメントする(アセンブラ) */
    printf("%d",TestValue);          /* 結果をプリント */
}
```

次に下のコマンドラインでコンパイルしてください。

```
tcc -s plusone
```

-Sオプションはソースファイルをコンパイルしてアセンブラコードに変換したら処理を停止することをTurbo Cに指示しています。この時点でディスク上にPLUSONE.ASMというファイルが作られているはずです。このファイルの内容は次のとおりです。

```
        ifndef  ??version
?debug  macro
        endm
        endif
        name    Plusone
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
DGROUP  GROUP   _DATA,_BSS
        ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP
_TEXT   ENDS
_DATA   SEGMENT WORD PUBLIC 'DATA'
_d@     label  BYTE
_d@w    label  WORD
_DATA   ENDS
_BSS    SEGMENT WORD PUBLIC 'BSS'
_d@     label  BYTE
_d@w    label  WORD
        ?debug C E90156E11009706C75736F6E652E63
        ?debug C E90009B9100F696E636C7564655C737464696F2E68
        ?debug C E90009B91000696E636C7564655C7374646172672E68
_BSS    ENDS
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
;       ?debug  L 3
_main   PROC    NEAR
        push    bp
        mov     bp,sp
        dec     sp
        dec     sp
;       ?debug  L 8
        lea     ax,WORD PTR [bp-2]
```

```
        push    ax
        mov     ax,OFFSET DGROUP:_s@
        push    ax
        call    NEAR PTR _scanf
        pop     cx
        pop     cx
;       ?debug  L 10
        inc     WORD PTR [bp-2]
        mov     ax,OFFSET DGROUP:_s@+3
        push    ax
        call    NEAR PTR _printf
        pop     cx
        pop     cx
@1:
;       ?debug  L 12
        mov     sp,bp
        pop     bp
        ret
_main   ENDP
_TEXT   ENDS
_DATA   SEGMENT WORD PUBLIC 'DATA'
_s@     label   BYTE
        db      37
        db      100
        db      0
        db      37
        db      100
        db      0
_DATA   ENDS
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
        EXTERN  _printf:NEAR
        EXTERN  _printf:NEAR
_TEXT   ENDS
        PUBLIC  _main
        END
```

（このプログラムを見れば,Turbo Cがインラインアセンブラをサポートしていることがいかに便利なことか，そのありがたみがわかると思います)。

この例では,次のコメント

```
;    ?debug  L 8
```

の後に**scanf**関数のアセンブラコードがあるのがわかります。またその後に続く

```
;     ?debug  L 9
      inc     WORD PTR [bp-2]
```

のコードは*TestValue*をインクリメントするインラインアセンブラの命令文に対応する命令です（**注意**：Turbo CではC変数の*TestValue*をアセンブラの対応アドレス［BP-2］へ自動変換します)。インラインアセンブラ命令の後は**printf**関数のアセンブラコードです。

ここでのポイントは,Turbo Cは**scanf**関数をアセンブラコードにコンパイルし,インラインアセンブラの命令文を直接アセンブラの出力ファイルに変換し,さらに**printf**関数をアセンブラコードにコンパイルすることです。この出力ファイルとして作られるファイルはアセンブラのソースファイルです。このファイルは,Turbo Assemblerを使ってアセンブルすることができます。

-**S**オプションを使わない場合は,Turbo Cは処理を停止せずに引き続きTurbo Assemblerを起動してPLUSONE.ASMをアセンブルし,次にTLINKを起動してオブジェクトファイルPLUSONE.OBJをリンクし,実行可能ファイルPLUSONE.EXEを作ります。Turbo Cでインラインアセンブラを使う場合は,通常この方法を採用します。-**S**オプションは,インラインアセンブラを使った場合にTurbo Cが行なう途中のアセンブリ処理をチェックするなど,補足的な目的にのみ使います。

-**S**オプションはプログラムコードをコンパイル,リンクして実行可能プログラムを作るために特に役に立つものではありませんが,インラインアセンブラコードの周辺の命令やTurbo Cが作った命令をチェックする際に便利です。また大切なことは,インラインアセンブラを使用したときは,インラインアセンブラコードが使用しているレジスタと，Turbo Cが生成したコードが使用しているレジスタが干渉していないかどうか確認するように習慣づけた方が良いかもしれません。インラインアセンブラコードからどんなコードが作られるのか詳しく知りたければ,アセンブル時に-**S**オプションを指定して，.ASMファイルの内容をチェックしてください。

インラインアセンブラモードの指定方法

通常Turbo CはCコードをコンパイルして直接オブジェクトコードを作ります。Turbo Cでインラインアセンブラモード(Cのソースをコンパイルしてアセンブラソースを作り,Turbo Assemblerを起動するモード）を指定するには次の3通りの方法があります。

-**B**コマンドラインオプションを付けると,Turbo Cはソースをコンパイルしてアセンブラコードを作ります。次にTurbo Assemblerを起動してこのコードをアセンブルさせます。

-**S**コマンドラインオプションを付けた場合は,Turbo Cはソースをコンパイルしてアセンブラコードを作ると一時的に処理を停止します。作られた.ASMファイルは個別にアセンブルしてCやアセンブラの任意のモジュールとリンクすることができます。一般的にプログラムのデバッグやチェック以外の目的では（-**B**オプションほど）-**S**オプションを使うことはありません。

#pragma指令を次のように使うと,

```
#pragma inline
```

-Bコマンドラインオプションを使ったのと同じ効果があります。つまりTurbo Cはソースをコンパイルしてアセンブラコードを作り，次にTurbo Assemblerを起動してこのコードをアセンブルさせます。Turbo Cが**#pragma inline**を見つけると，コンパイル処理はアセンブラ出力モードになります。**#pragma inline**はCのソースコードの先頭にできるだけ近いところに置くのがよいでしょう。なぜなら**#pragma inline**より前にソースコードがあると，そのソースは2回コンパイルされてしまいます(はじめは通常のC-オブジェクトモードでコンパイルされ，次にC-アセンブラモードで再びコンパイルされます)。こうなっても害はありませんが，時間の無駄です。

インラインアセンブラコードを使ったにもかかわらず，上のどのオプション（**-B**, **-S**）も**#pragma inline**も指定しないでコンパイルすると，次のような警告メッセージが出力されます。

```
Warning test.c 6: Restarting compile using assembly in function main
```

この場合コンパイラはインラインアセンブラコードの位置に**#pragma inline**があるものと仮定して，アセンブラ出力モードに切り換えてコンパイルを再開します。インラインアセンブラを使う場合は，**-B**オプションや**#pragma inline**を指定して，このメッセージが出ないようにしてください。上記のようにインラインアセンブラコード検出時にコンパイルを再開すると，コンパイル速度がやや遅くなります。

Turbo Assemblerを起動してインラインアセンブラコードをアセンブルする

Turbo CがTurbo Assemblerを起動するには,Turbo Cはその**存在場所を知る**必要があります。この検索方法はTurbo Cのバージョンによって異なります。

Turbo Cの2.0以降のバージョンでは，カレントディレクトリまたはDOSのPATH環境変数で指定されたディレクトリ内のTASM.EXEというファイルをTurbo Assemblerプログラムファイルとして認識します。通常コマンドラインプロンプトから

```
TASM
```

というコマンドを入力してTurbo Assemblerを開始する場合と同じ環境条件で，Turbo CはTurbo Assemblerを起動します。したがってTurbo Assemblerがカレントディレクトリ内またはコマンドのサーチパス内のどこかにあれば，Turbo CはTurbo Assemblerを起動してインラインアセンブラをアセンブルします。

Turbo Cのバージョン1.0と1.5は動作が若干違います。これらのバージョンはTurbo Assemblerができる前に作られたため，マイクロソフト社のマクロアセンブラMASMを起

動してインラインアセンブラをアセンブルします。つまりこの2つのバージョンはカレントディレクトリ内またはコマンドのサーチパス内を検索して，プログラムファイル(TASM.EXEではなく）MASM.EXEを起動します。Turbo Assemblerは起動されません。

注意：この2つのバージョンにパッチをあてて，TASMを使うようにすることができます。Turbo Assemblerのマスターディスク内のREAD.MEというファイルにその説明が入っていますので参照してください。

Turbo Cはインラインアセンブラコードをどこへストアするか

インラインアセンブラコードはTurbo Cのコードセグメントまたはデータセグメント内にストアされます。Turbo Cは関数内にあるインラインアセンブラの命令文をアセンブルするとTurbo Cのコードセグメント内にストアします。また関数の外にあるインラインアセンブラの命令文をアセンブルすると，Turbo Cのデータセグメント内にストアします。次に例を示します。

```
/* 平方値のテーブル */
asm  SquareLookUpTable  label  word;
asm  dw  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100;

/* 0～10の平方値を得る関数 */
int LookUpSquare(int Value)
{
    asm  mov  bx,Value;                      /* 2乗する数値をロードする */
    asm  shl  bx,1;                          /* テーブル内のワードサイズの要素
                                                を参照するために2倍する */
    asm  mov  ax,[SquareLoopUpTable+bx];     /* 平方値をロードする */
    return(_AX);                             /* 結果を返す */
}
```

この例では，*SquareLookUpTable*のデータはTurbo Cのデータセグメント内にストアされ，関数*LookUpSquare*内のインラインアセンブラコードはTurbo Cのコードセグメント内にストアされます。データもコードセグメント内にストアすることは可能です。そのために先ほどの例を書き直すと次のようになります。この例では，*SquareLookUpTable*のデータはTurbo Cのコードセグメント内にストアされます。

```
/* 0～10の平方値を得る関数 */
int LookUpSquare(int Value)
{
    asm  jmp  SkipAroundData                  /* データテーブルを飛び越す */

    /* 平方値のテーブル */
    asm  SquareLookUpTable  label  word;
    asm  dw  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100;

SkipAroundData:
    asm  mov  bx,Value;                       /* 2乗する数値をロードする */
    asm  shl  bx,1;                           /* テーブル内のワードサイズの要素
                                                 を参照するために2倍する */
    asm  mov  ax,[SquareLoopUpTable+bx];      /* 平方値をロードする */
    return(_AX);                              /* 結果を返す */
}
```

*SquareLookUpTable*のデータがTurbo Cのコードセグメント内にあるため，データを読むにはCS：セグメントオーバライドプレフィクスが必要であるように思われます。しかし実際は*SquareLookUpTable*をアクセスするコードは,CS：プレフィクスが付けられてアセンブルされます。つまりTurbo Cがアセンブラコードを作る際は，*SquareLookUpTable*がどのセグメント内にあるかがTurbo Assemblerにわかるようにするわけです。それに応じてTurbo Assemblerがセグメントオーバライドプレフィクスを付けます。

80186と80286の命令には-1スイッチを使う

shr ax, 3またはpush 1などの80186プロセッサ固有のアセンブラ命令を使う場合は，次のように-**1**コマンドラインオプションを使ってTurbo Cを開始してください。

```
tcc -1 -b heapmgr
```

ただしHEAPMGR.Cは,80186固有のインラインアセンブラ命令を含むプログラムファイルです。

-**1**コマンドラインオプションの基本機能は,コンパイル時に80186の命令のフルセットを利用することをTurbo Cに指示することです。またこのオプションは,アセンブラの出力ファイルの先頭に**.186**疑似命令を挿入することを指定します。これによって80186の命令のフルセットをアセンブルすることをTurbo Assemblerに指示します。**.186**疑似命令を使わないと,Turbo Assemblerは80186固有のインラインアセンブラ命令をエラーとして通知します。Turbo Cに80186命令のフルセットの使用を指示することなく80186命令をアセンブルしたい場合は,80186のインラインアセンブラ命令が使われたTurbo Cモジュールの先頭に次の行を挿入してください。

```
asm  .186;
```

この行はアセンブラファイルに入り,80186命令のアセンブルをTurbo Assemblerに指示します。

Turbo Cは, 80286, 80386, 80287, 80387プロセッサを標準ではサポートしていませんが, 80286, 80287, 80386, 80387をサポートするインラインアセンブラは上記と同じような方法で使用可能となります。つまりasmキーワードと**.286, .286C, .286P, .386, .386C, .386P, .287, .387**などのTurbo Assemblerの疑似命令を使えばよいのです。

asm .186;という疑似命令はインラインアセンブラを使ううえでは重要なものです。セグメント疑似命令,イコール定義,マクロ命令などを**asm**プレフィクスと一緒に使うことによって,アセンブラ命令をアセンブラファイルにストアすることができます。

インラインアセンブラ命令文の形式

インラインアセンブラの命令文は通常のアセンブラ命令と非常によく似ていますが,若干の違いもあります。インラインアセンブラの命令文の一般形式は次のようになります。

```
asm  [<ラベル>] <機械命令/疑似命令> <オペランド> <; または改行>
```

- 各インラインアセンブラの命令文の先頭には**asm**キーワードを付けなければなりません。
- **<ラベル>**はアセンブラで使用できるラベルです。アセンブラの場合と同様, [] はラベルがオプション (任意指定項目) であることを示します (Cとアセンブラのラベルの違いについては少し後の「メモリオペランドとアドレスオペランドの制限」の節を参照してください)。
- **<機械命令/疑似命令>**はアセンブラの機械命令または疑似命令です。
- **<オペランド>**はそれぞれの命令に合ったオペランドです。ある制限内でC言語の定数,変数,ラベルなどを参照することができます。この制限については298ページのセクション「メモリオペランドとアドレスオペランドの制限」を参照してください。
- **<;または改行>**はセミコロンまたは改行のどちらかを表わします。**asm**文の終わりを示します。

注意：**ラベル**に関する重要事項は, 少し後の「メモリオペランドとアドレスオペランドの制限」の節を参照してください。

インラインアセンブラのセミコロン

C言語のプログラマの誰もが気付くことですが，インラインアセンブラの命令文の終わりにはC言語の文で付けたようなセミコロンは必要ありません。もちろんインラインアセンブラの命令文の終わりにセミコロンを付けてもかまいませんが，行末に達すれば自動的に文の終わりとみなされます（プログラムの見やすさという点からいうとあまりよい習慣とはいえませんが）1行に複数のインラインアセンブラの命令文を記述する場合を除いて，通常はセミコロンは付けても付けなくてもよいのです。これはC言語の規定に合わないように見えますが，特定のUNIX用コンパイラの規定に合わせているだけです。

インラインアセンブラのコメント

インラインアセンブラの命令文の形式に関するこれまでの説明には大切なことが1つ欠けています。それはコメントフィールドです。インラインアセンブラの命令文の最後にセミコロンを付けると，Cの文の場合と同様に文の最後とみなされます。つまりインラインアセンブラの命令文では，（通常のアセンブラコードとは異なり）セミコロンはコメントフィールドの開始としては使えません。

ではインラインアセンブラでコメントを付けるにはどうすればよいのでしょう？実はC言語のコメント形式で付けます。これは少し妙なことに思われるかもしれませんが，理由があります。それはCのプリプロセッサは他のCの文とともにインラインアセンブラの命令文を処理するからです。この方法に従えばCプログラム内にインラインアセンブラがある場合でも，一貫したスタイルでコメントを付けることができます。またCとインラインアセンブラの両コード内でC定義シンボルを利用することができます。次に例を示します。

```
     .
     .
     .
#define CONSTANT 51
     int i;
     .
     .
     .
     i = CONSTANT;                    /* iに定数をセット */
     asm  sub  WORD PTR i,CONSTANT;   /* iから定数を引く*/
     .
     .
     .
```

この例ではCで定義されたシンボル*CONSTANT*を，Cとインラインアセンブラの両方が

利用しています。*i*は0になります。

この例はオペランドフィールドからC定義のシンボル名だけでなく，Cの変数を直接参照できるというインラインアセンブラの優れた特長を示すものです。この章で後述しますが，通常アセンブラでC変数をアクセスすることはめんどうな作業です。しかしインラインアセンブラを使えばC変数を簡単に参照することができます。この点が，ほとんどのアプリケーションでアセンブラとCを組み合わせて使うことが好ましい主な理由です。

構造体と共用体の要素のアクセス

インラインアセンブラから構造体の要素を直接参照することができます。次に例を示します。

```
          .
          .
          .
  struct Student {
          char Teacher[30];
          int Grade;
  } JohnQPublic;
          .
          .
          .
          asm  mov  ax,JohnQPublic.Grade;
          .
          .
          .
```

この例は*Student*型の*JohnQPublic*という名の構造体のメンバ*Grade*の内容をAXにロードします。

インラインアセンブラコードを使えば，ベースレジスタまたはインデックスレジスタのオフセットで構造体の要素にアクセスすることもできます。次に例を示します。

```
  .
  .
  .
  asm  mov  bx,OFFSET JohnQPublic;
  asm  mov  ax,[bx],Grade;
  .
  .
  .
```

この例も構造体*JohnQPublic*のメンバ*Grade*の内容をAXにロードします。*Grade*は*Student*構造体のオフセット30にあるため，この例は次のように書き直すことができます。

```
.
.
.
asm  mov  bx,OFFSET JohnQPublic;
asm  mov  ax,[bx]+30
.
.
.
```

ポインタレジスタのオフセットで構造体要素にアクセスできるのは非常にパワフルな機能です。インラインアセンブラコードで構造体の配列や構造体に渡されたポインタを扱うことができます。

複数の構造体が同一のメンバ名をもっていて，インラインアセンブラでこれらの構造体にアクセスする場合は，

```
asm  mov  bx,[dl].(struct tm) tm_hour > alt
```

を挿入してください。次にその例を示します。

```
     .
     .
     .
struct Student {
     char Teacher[30];
     int Grade;
} JohnQPublic;
     .
     .
     .
struct Teacher {
     int Grade;
     long Income;
};
     .
     .
     .
asm  mov  ax,JohnQPublic.(struct Student) Grade
     .
     .
     .
```

インラインアセンブラのプログラム例

これまでに示したインラインアセンブラを使ったさまざまなプログラム例は，どれも実際には動かないプログラムです。この節では，インラインアセンブラを使ってテキストの大文字への変換処理の速度を大幅に上げたプログラムを紹介します。この節のプログラム

例を見れば，インラインアセンブラの能力が理解できます。インラインアセンブラでプログラムを作る際に参考にしてください。

ここでサンプルプログラムを使ってプログラミング上の問題点をチェックしておきましょう。与えられた文字列の小文字を大文字に変換しながら他の文字列に転送する，*StringToUpper*という名前の関数を考えます。この関数が任意のメモリモデルの文字列を処理できるようにするための1つの方法として，far文字列のポインタを関数に渡す方法があります。near文字列のポインタはfar文字列のポインタに型キャストできますが，逆方向に型キャストできるとは限らないためです。

ここで8086プロセッサの性能の問題に突き当たります。Turbo Cはfarポインタを扱うことができますが，nearポインタを扱うのに較べると処理速度がかなり落ちます。これはTurbo Cの欠点ではなく，8086で高級言語を処理する際の避けられない現象です。

一方，文字列とfarポインタの処理はアセンブラが得意とする分野の1つです。インラインアセンブラを使ってfarポインタの処理と文字列の転送を行ない，その他の処理はすべてTurbo Cで記述するのが合理的です。次に例としてプログラムSTRINGUP.Cを示します。

```
/* StfingToUpperの使い方を示すプログラム
   機能： StringToUpperをコールしてTestStringを大文字に変換し，
   UpperCaseStringにストアする． 次にUpperCaseStringとその長さをプリントする． */

#pragma inline
#include <stdio.h>

/* StringToUpperの関数プロトタイプ */
extern unsigned int StringToUpper(
unsigned char far * DestFarString,
unsigned char far * SourceFarString);

#define MAX_STRING_LENGTH 100

char *TestString = "This Started Out As Lowercase!";

char UpperCaseString[MAX_STRING_LENGTH];

main()
{
   unsinged int StringLength;

   /* TestStringの大文字バージョンをUpperCaseStringに転送する */
   StringLength = StringToUpper(UpperCaseString, TestString);

   /* 変換結果を表示 */
   printf("Original string:¥n%s¥n¥n",TestString);
   printf("Uppercase string:¥n%s¥n¥n",UpperCaseString);
   printf("Number of characters:¥n%s¥n¥n",StringLength);
}

/* far文字列から別のfar文字列へ大文字の高速変換を実行する関数
```

```
   入力:
      DestFarString   - 大文字の文字列をストアする配列(0で終わる)
      SourceFarString - 大文字へ変換する文字列(0で終わる)

   戻り値:
      ソース文字列の長さ(文字数) (終端の0は除く)  */
unsigned int StringToUpper(unsigned char far * DestFarString,
                           unsigned char far * SourceFarString)
{
   unsigned int  CharacterCount;

   #define LOWER_CASE_A 'a'
   #define LOWER_CASE_Z 'z'
      asm ADJUST_VALUE  EQU  20h;     /* 小文字を大文字に変換するために
                                         マイナスする値 */


      asm  cld;
      asm  push ds;                   /* Cのデータセグメントを退避 */
      asm  lds  si,SourceFarString;   /* ソース文字列のfarポインタをロード */
      asm  les  si,DestFarString;     /* デスティネーション文字列のfarポインタ
                                         をロード */
      CharacterCount = 0              /* 文字列カウンタ */
   StringToUpperLoop:
      asm  lodsb;                     /* 次の文字をロード */
      asm  cmp  al,LOWER_CASE_A;      /* 'a'より小さければ小文字でない */
      asm  jb   SaveCharacter;
      asm  cmp  al,LOWER_CASE_Z;      /* 'z'より大きければ小文字でない */
      asm  ja   SaveCharacter;
      asm  sub  al,ADJUST_VALUE;      /* 小文字なので大文字に変換 */
   SaveCharacter:
      asm  stosb;                     /* 文字をストア */
      CharacterCount++;               /* 文字をカウント */
      asm  and  al,al;                /* 終端の0か? */
      asm  jnz  StringToUpperLoop;    /* No...次の文字があれば処理する */
      CharacterCount--;               /* Yes...終端の0はカウントしない */
      asm  pop  ds;                   /* Cのデータセグメントを復帰 */
      return(CharacterCount);
}
```

このプログラムを実行すると次のメッセージが表示されます。

```
Original string:
This Started Out As Lowercase!

Uppercase string:
THIS STARTED OUT AS UPPERCASE!

Number of characters: 30
```

すべての小文字が大文字に変換されたことがわかります。

STRINGUP.Cの中心は*StringToUpper*という関数です。この関数は文字列の転送と大文

字への変換処理をすべて行ないます。*StringToUpper*はCとインラインアセンブラの両方で書かれており，パラメータとして2つのfarポインタを受け取ります。1つのfarポインタはテキストを含む文字列（ソース文字列）を指し，もう1つのfarポインタは他の格納域（デスティネーション格納域）を指します。ソース文字列のテキストの小文字が大文字に変換されてデスティネーション格納域へ転送されます。関数の宣言とパラメータの定義はすべてCで行なわれ，プログラムの先頭に*StringToUpper*の関数 プロトタイプがあります。メインプログラムはあたかも*StringToUpper*がすべてCだけで記述されているように*StringToUpper*を呼び出しています。*StringToUpper*の一部にインラインアセンブラが使われています。Turbo Cプログラミングのすべての利点がこのプログラムに示されています。

*StringToUpper*の本体はCとインラインアセンブラが混在しています。アセンブラはソース文字列からの文字の読み込み，文字のチェック，大文字への変換，デスティネーション文字列への文字の転送などに使われています。インラインアセンブラを利用すれば，ストリング命令の**LODSB**と**STOSB**を使って文字の読み書きができます。

*StringToUpper*を作る際は，Turbo Cのデータセグメント内のデータをアクセスする必要がないことがわかっているため，関数の先頭でDSをプッシュし，次にDSにソース文字列へのポインタをセットしています（以後DSを変更しません）。インラインアセンブラがCに勝っている点の1つに，関数の先頭でfarポインタを1回だけセットでき，以後それを変更しないことです。これに対してTurbo Cなどの高級言語では実行するごとにfarポインタをセットします。インラインアセンブラを使えばfarポインタのセットが1回で済むため，*StringToUpper*はnear文字列と同じように，高速でfar文字列を処理することができます。

*StringToUpper*に関する別の興味深い点は，Cとアセンブラの文が混在していることです。*LOWER_CASE_A*と*LOWER_CASE_Z*の定義には**#define**を使っており，*ADJUST_VALUE*の定義にはEQU疑似命令を使っています。しかしインラインアセンブラコードはこれら3つのシンボルを同じように扱っています。C定義シンボルの置き換えはTurbo Cのプリプロセッサが行ない，*ADJUST_VALUE*の置き換えはTurbo Assemblerが行ないます。インラインアセンブラコードはその両方を扱うことができます。

*StringToUpper*内にはあちこちに*CharacterCount*を扱うCの文があります。このことからCコードとインラインアセンブラコードが混在できることがわかります。CXまたはDXなどのレジスタ内のインラインアセンブラコードの組み合わせによって，*CharacterCount*の処理を簡単に直接行なうこともできます。そうすれば*StringToUpper*の実行速度はもっと上がるはずです。

Cコードとインラインアセンブラコードを自由に混在させることができるのは便利です。しかしその反面，Turbo Cがインラインアセンブラの命令文からどのようなコードを作るのかを正しく把握していないと危険です。Turbo Cの-Sコンパイラオプションを使えば，Cとインラインアセンブラを混在したときにどのように処理されるかを知ることができま

す。たとえば-Sコンパイラオプションを使ってSTRINGUP.Cをコンパイルして出力ファイルSTRINGUP.ASMの内容をチェックすれば,*StringToUpper*内のCコードとインラインアセンブラコードがどのように統合されるのかを正しく理解することができます。

STRINGUP.Cはインラインアセンブラをうまく使えば,優れた効果をもたらすことを示すプログラム例です。*StringToUpper*では15行のインラインアセンブラコードを使っており，その文字列処理の速度はC言語だけで記述した場合の約2倍になっています。

インラインアセンブラの制限

インラインアセンブラの使用法にはほとんど制限がありません。インラインアセンブラの命令文はそのままTurbo Assemblerに渡されるだけだからです。ただしメモリオペランドとアドレスオペランドを使う場合は気をつけなければならない制限があります。またレジスタの使用法に関していくつかの制限があります。たとえばインラインアセンブラの命令文で使われるCの自動変数では，デフォルトサイズ調整は行なわれません。

メモリオペランドとアドレスオペランドの制限

Turbo Cがインラインアセンブラの命令文に変更を施す場合があります。Turbo Cは,変数名やジャンプ先などのメモリ参照およびメモリアドレス参照の形式をC言語形式からアセンブラ形式に変換します。これらの変更によって2点だけ制限が加わります。インラインアセンブラでは，ジャンプ命令はCラベルしか参照できず，それ以外の命令はCラベルを参照できません。たとえば次の例は正しくコンパイルされますが,

```
.
.
.
asm  jz  NoDec;
asm  dec  cx;
NoDec:
.
.
.
```

次の例は正しくコンパイルされません。

```
  .
  .
  .
  asm  jz  NoDec;
  asm  dec  cx;
  asm  NoDec:
  .
  .
  .
```

同様にインラインアセンブラのジャンプ命令のオペランドに関数名を記述することはできません。

インラインアセンブラのジャンプ以外の命令はCラベルを参照することはできません。次の例は正しくコンパイルされますが,

```
  .
  .
  .
  asm  BaseValue  db  '0';
  .
  .
  .
  asm  mov  al,BYTE PTR BaseValue;
  .
  .
  .
```

次の例は正しくコンパイルされません。

```
      .
      .
      .
BaseValue:
      asm  db  '0';
      .
      .
      .
      asm  mov  al,BYTE PTR BaseValue;
      .
      .
      .
```

コールはジャンプとは異なります。インラインアセンブラのコール命令のオペランドには, Cの関数名やアセンブララベルを使うことができます。インラインアセンブラの命令でCの関数名を参照する場合は, 命令の前に下線記号のプレフィクスを付ける必要があります。詳細は少し後の「下線記号」の節を参照してください。

インラインアセンブラには自動変数のデフォルトサイズ調整がない

Turbo Cがインラインアセンブラの命令文の自動変数への参照を［*BP-02*］などのオペランドに置き換える場合は，変更後の文に**WORD PTR**や**BYTE PTR**などのサイズ演算子を付けません。次に例を示します。

```
.
.
.
int i;
.
.
.
asm  mov  ax,i;
.
.
.
```

このプログラムをコンパイルすると次のアセンブラコードに変換されます。

```
mov  ax,[bp-02]
```

この例は問題ありません。オペランドにAXを使えばTurbo Assemblerは16ビットメモリの参照として認識します。さらにサイズ演算子がないためインラインアセンブラでオペランドサイズを柔軟に指定することができます。次に例を示します。

```
.
.
.
int i;
.
.
.
asm  mov  i,0;
asm  inc  i;
.
.
.
```

このプログラムをコンパイルすると次のアセンブラコードに変換されます。

```
mov  [bp-02],0
inc  [bp-02]
```

この2つの命令はどちらも本来のサイズを持っておらず,Turbo Assemblerはこれらをアセンブルすることができません。したがってソースまたはデスティネーションオペランド

にレジスタを指定しないでアセンブラの自動変数を参照する場合は,サイズ演算子を必らず指定してください。上の例にサイズ演算子を指定すると次のようになります。

```
.
.
.
int i;
.
.
.
asm  mov  WORD PTR i,0;
asm  inc  BYTE PTR i;
.
.
.
```

レジスタ退避の必要性

インラインアセンブラコードの最後では,レジスタ(BP, SP, CS, DS, SS)の値をインラインアセンブラコードの先頭における値に戻しておく**必要があります**。このルールを守らないと,しばしばプログラム暴走またはシステムリブートの原因となります。AX, BX, CX, DX, SI, DI, ES, フラグはインラインアセンブラコードで自由に変更することができます。

関数の呼び出しとレジスタ変数の退避

Turbo Cでは，SIとDI(レジスタ変数として使われます)が関数の呼び出しによって破壊されないことが条件です。インラインアセンブラの命令文でSIやDIを使う場合は，幸いこれらを意識的に退避する必要はありません。Turbo Cはこれらのレジスタを関数の先頭で退避し，関数の最後で復帰してくれます。

内部レジスタ変数の消滅

レジスタ変数はSIやDIにストアされるため，同一モジュール内のレジスタ変数と(SIやDIを使う)インラインアセンブラコードの間で衝突が起こる可能性があります。しかしTurbo Cではこの問題に対して対策が施されています。つまりインラインアセンブラコードでSIやDIを使った場合は，これらのレジスタにレジスタ変数をストアするすることはできません。

Turbo Cのバージョン1.0ではこの問題(レジスタ変数とインラインアセンブラコードの間での衝突)に対して上記のような対策が施されていません。したがってバージョン1.0で

インラインアセンブラコードでSIやDIを使う場合は，あらかじめこれらのレジスタを意識的に退避してください。またはTurbo Cの最新バージョンに更新してください。

インラインアセンブラと純粋なCの欠点

これまではインラインアセンブラの機能を解説し，その長所を学んできました。インラインアセンブラは優れた機能を持っていますが，同時にいくつかの欠点もあります。この節ではインラインアセンブラの短所を学習し，どのような場合にプログラム内でインラインアセンブラを使ったらよいのかが理解できます。

移植性（可搬性）と保守性が悪くなる

インラインアセンブラが非常に優れている点は，8086プロセッサに対して直接プログラミングすることができることです。しかしインラインアセンブラを使うと，C言語の基本的な特長である「移植性」は悪くなります。この場合，プログラムをそのまま他のプロセッサまたはCコンパイラに移植することはほぼ確実にできなくなります。

またインラインアセンブラにはCのような明瞭性や統一のとれた書式がなく，ほとんどの場合構造化されることはありません。したがってインラインアセンブラを使うとC言語だけの場合より読みにくくなり保守性も悪くなります。

インラインアセンブラを使う場合は，インラインアセンブラプログラムはモジュール化して切り離し，多くのコメントを付けて構造化することを習慣にすべきです。このようにすればプログラムの保守が楽になります。またプログラムを他の環境に移植する必要が生じた場合，インラインアセンブラプログラム部分をCに書き直すことも比較的容易に行なうことができます。

コンパイル速度の低下

インラインアセンブラを含むCプログラムのコンパイル速度は，純粋なCプログラムのコンパイル速度よりかなり遅くなります。第1の理由はインラインアセンブラコードは最初はTurbo Cによって，次にTurbo Assemblerによって，合計2回コンパイルされる必要があるためです。-Bオプション，-Sオプション，**#pragma inline**のいずれも使わずにTurbo Cがコンパイルを開始した場合は，インラインアセンブラのコンパイルに要する時間はさらに長くなります。しかしTurbo Assemblerの処理速度は以前のアセンブラより非常に速いため，（インラインアセンブラを含むモジュールのコンパイル速度が落ちるとはいえ）以前に比

べたら問題にならない程度です。

TCCのみで使用可能

すでに述べたようにインラインアセンブラはTCC.EXE（Turbo Cのコマンドラインバージョン）にのみ備っている機能です。TC.EXE（Turbo Cの統合開発環境バージョン）はインラインアセンブラをサポートしていません。

最適化が犠牲となる

インラインアセンブラの命令文はCプログラム内のどこにでも挿入できるため,インラインアセンブラを使った場合はTurbo Cのプログラム最適化処理の一部が犠牲になります。したがってプログラマ自身がこの犠牲をある程度補う必要があります。たとえばDSレジスタの退避し忘れやメモリの不正領域へのデータ書き込みなどの不注意は避けなければなりません。

一方インラインアセンブラでプログラムを作る場合は,プログラマはTurbo Cのすべてのルールに従う必要はありません。もし従わなければならないとしたら,Cのかわりにインラインアセンブラを使う利点は減少してしまいます。関数内にインラインアセンブラを入れると関数の最適化処理の一部が取り消されますが,それさえがまんすればインラインアセンブラを自由に記述することができます。またこの場合はジャンプの最適化処理の一部も取り消され,インラインアセンブラでSIとDIを使った場合はレジスタ変数は使用不能となります。しかしプログラムの性能を最大限に引き出すためにインラインアセンブラを使う限り,このように最適化処理の一部が取り消されても決して無駄にはなりません。

インラインアセンブラを使って高速・コンパクトなプログラムを作りたい場合は,関数内にCとインラインアセンブラを混在させずにインラインアセンブラのみで記述するのがよいでしょう。そうすればインラインアセンブラ関数とC関数のそれぞれ独自の最適化が図られ,制限のない最適なプログラムを作ることが可能となります。

エラートレースの制限

Turbo Cはインラインアセンブラの命令文のエラーチェックはほとんど行ないません。そのかわりほとんどの場合Turbo Assemblerがインラインアセンブラ内のエラーを検出します。エラー メッセージと行番号はCプログラム自身ではなくTurbo　Cが出力する.ASMファイルに基づいているため,Turbo Assemblerが出力したエラー メッセージをオリジナルのCソースプログラムと関連させることは困難です。

たとえばインラインアセンブラを含むCプログラムTEST.Cをコンパイルする際に,Turbo Assemblerは不正オペランドサイズが行23にあることを通知するとします。この"23"とは,Turbo Cが作りTurbo Assemblerがアセンブルする中間アセンブラファイルTEST.ASM内のエラーが発生した行の番号です。この行がTEST.Cのどの行に該当するのかは,プログラマが判断しなければなりません。

このような場合は,中間.ASMファイル内のエラーが発生した行をまず見つけるのがよいでしょう。Turbo Assemblerがアセンブルエラーを通知した場合は,Turbo Cはこのファイルをディスク上にそのまま残しています。.ASMファイルには特別なコメントが入れられています。このコメントはアセンブラ文の各ブロックがCのソースファイルのどの行から作られたものかを示します。たとえばコメントは,

```
; ?Debug L 15
```

このコメント以降の行がCのソースファイルの15行目から作られたコードであることを示します。.ASMファイル内のエラーを発生した行番号がわかれば,この行番号コメントを使ってCのソースファイル内でのエラー発生箇所をたどることができます。

デバッグの制限

バージョン1.5およびそれ以前のTurbo Cでは,インラインアセンブラを含むモジュールのソースレベルのデバッグ情報(デバッグ用にCのソースコードを示すために必要な情報)は出力しません。インラインアセンブラを使った場合は,Turbo Cのバージョン1.5とそれ以前のバージョンではアセンブラコードを出力するだけでデバッグ情報は出力しません。したがってインラインアセンブラを含むCのモジュールに対してはソースレベルのデバッグ機能はなく,アセンブラレベルのデバッグ機能のみがあります。

Turbo Cの2.0以降のバージョンとTurbo Debuggerを組み合わせてインラインアセンブラを含むモジュールのデバッグを行なった場合は,Turbo Debuggerの特殊機能である最新のソースレベルのデバッグ機能を利用することができます(もちろん純粋なCプログラムのデバッグにも利用できます)。

最初はCだけを使い,最終版にはインラインアセンブラも使う

これまで説明したインラインアセンブラの短所を考え合わせると,インラインアセンブラはできるだけひかえめに使うのがよいという結論に達しかねません。しかしそうではありません。インラインアセンブラのキーポイントは,プログラム開発過程の適切な場所で使うことです。

インラインアセンブラの使用が裏目にでる原因は,ほとんどの場合次の一点に絞ることができます。それはインラインアセンブラを使うと,エディット,コンパイル,デバッグのサイクルが著しく遅くなることです。コンパイル速度が低下すること,統合インターフェース環境が使えなくなること,コンパイルエラーの発見が困難になることは,インラインアセンブラを含むプログラムを作るのは純粋なCプログラムを作るより多くの時間を費やすことを意味します。しかしインラインアセンブラはその使用法が適切であれば,プログラムの性能が飛躍的に向上します。ではインラインアセンブラを適切に使うにはどうしたらよいのでしょうか?

その答えは簡単です。最初はプログラムのすべての部分をCだけで記述します。そうすればTC.EXEが持っている優れた開発環境をフルに利用することができます。プログラムのデバッグが済んで,すべての機能が問題なく動作するようになったら,TCC.EXEに切り替えてプログラム内の必要部分をインラインアセンブラで書き直します。このアプローチによって効率のよいプログラム開発とデバッグが可能となります。そしてプログラムの改版時には,プログラムの必要部分を分離してその部分だけを改良することができます。

Turbo CからTurbo Assembler関数を呼び出す

従来, Cプログラムとアセンブラのプログラムを組み合わせる場合は, 次の手順に従っていました。

1. Cとアセンブラのプログラムを別々に作る。
2. Cモジュールをコンパイルし,アセンブラモジュールをアセンブルする。
3. それぞれのモジュールをリンクする。

この方法によってTurbo CのモジュールをTurbo Assemblerのモジュールとリンクすることができます。図6.3にこのプロセスを示します。

このようにCとアセンブラのそれぞれのソースファイルから実行可能ファイルを作るプロセスは,次のコマンドラインで実行することができます。

```
tcc filenaml filenam2.asm
```

このコマンドラインは,はじめにFILENAM1.CをコンパイルしてFILENAM1.OBJを作り,Turbo Assemblerを起動してFILENAM2.ASMをアセンブルしてFILENAM2.OBJを作ります。最後にTLINKを起動してFILENAM1.OBJとFILENAM2.OBJをリンクしてFILENAM1.EXEを作ります。

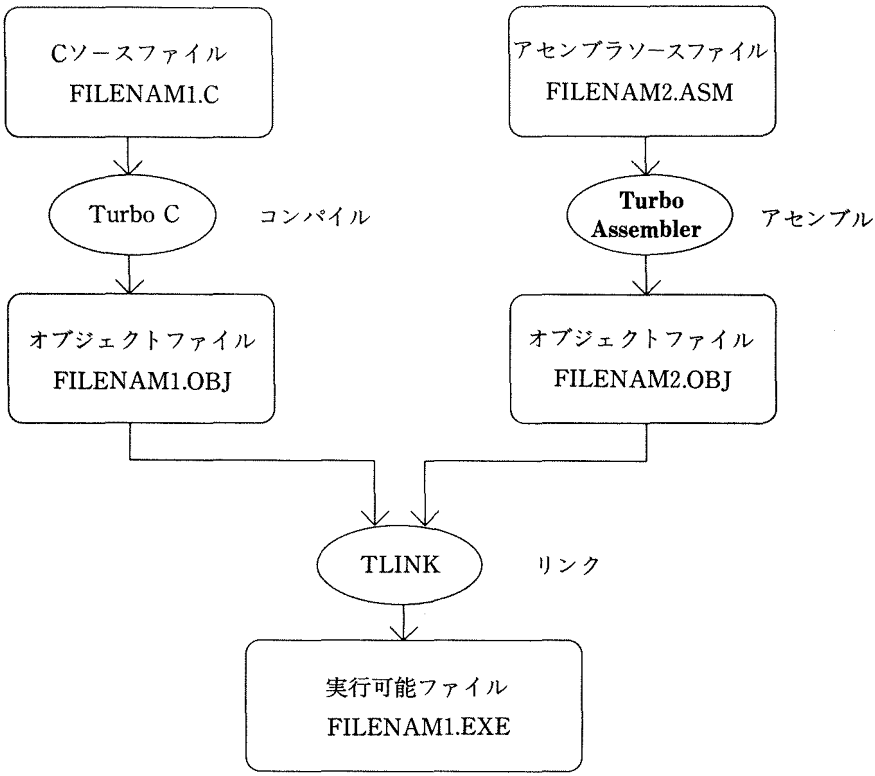


図6.3 Turbo C, Turbo Assembler, TLINKを使った コンパイル,アセンブル,リンクのサイクル

このような個別コンパイル（別々にオブジェクトファイルを作ってリンクする方法）は，可変サイズのアセンブラコードを含むプログラムに対してたいへん便利です。この方法をとればTurbo Assemblerの能力をフルに発揮させることができます。また**asm**キーワード，余計なコンパイル時間，インラインアセンブラのCに関連するオーバヘッドなどがない純粋なアセンブラ環境でアセンブラプログラミングを行なうことができます。

しかし個別コンパイルを行なうには多少の犠牲を払う必要があります。アセンブラプログラマはCとアセンブラコードのインターフェースの詳細に精通していなければなりません。Turbo Cがセグメント指定，パラメータの受け渡し，C変数の参照，レジスタ変数の退避，およびインラインアセンブラに対する同様な処理を扱うところでは，個別にコンパイルされたアセンブラ関数も同様にこれらすべての作業を行なう必要があります。

Turbo CとTurbo Assemblerのインターフェースには2つの側面があります。第1にCとアセンブラのプログラムの各部を適切にリンクし，各部の関数と変数がプログラムの他の部分から必要に応じて利用できなければなりません。第2にアセンブラプログラムはC形式の関数呼び出しを適切に扱えなければなりません。これには渡されたパラメータのアクセス，値のリターンなども含まれます。またレジスタ退避ルールにも従わなくてはなりません。

はじめにTurbo CとTurbo Assemblerのリンクのルールから説明します。

Turbo CとTurbo Assemblerのインターフェースの概要

Turbo CモジュールとTurbo Assemblerモジュールをリンクするには，次の3つの条件を満足しなくてはなりません。

Turbo AssemblerモジュールがTurbo Cと互換性のあるセグメント命名規定を使っていること。

Turbo CとTurbo Assemblerのモジュールは適切な関数と変数名を共用していること。しかもTurbo Cが利用できる形態で共用していること。

TLINKを使ってモジュールをリンクして実行可能プログラムを作ること。

この説明からはTurbo Assemblerモジュールが行なうことは何もわかりません。ここで説明したことは，Cと互換性のあるTurbo Assemblerの関数を作るための前提にすぎません。

メモリモデルとセグメント

アセンブラ関数（アセンブラで作成した関数）がCプログラムからコールされるための条件は，その関数がCプログラムが同じメモリモデルを使い，Cと互換性のあるコードセグメントを使うことです。同様にCプログラムからアセンブラモジュール内で定義されたデータをアクセスするためには（またはアセンブラプログラムからCデータをアクセスするためには），アセンブラプログラムはCのデータセグメント命名規定に従わなくてはなりません。

メモリモデルとセグメントをアセンブラで扱うのは非常に複雑です。しかし幸いTurbo Cと互換性のあるメモリモデルとセグメントの組み込み作業はすべてTurbo Assemblerが行ないます。このとき使われるのが簡略セグメント疑似命令です（簡略セグメント疑似命令の詳細は，第4章の「標準セグメント疑似命令」の節を参照してください）。

簡略セグメント疑似命令とTurbo C

DOSSEG疑似命令は，インテル社のセグメントオーダ規定に従ってセグメントの順番を決めることをTurbo Assemblerに指示します。インテル社のセグメントオーダ規定は，Turbo C (およびマイクロソフト社の言語その他のポピュラーなプログラミング言語) によって使われています。

.MODEL疑似命令は，簡略セグメント疑似命令によって作られるセグメントを指定のメモリモデル（タイニィ，スモール，コンパクト，ミディアム，ラージ，ヒュージ）で使うことをTurbo Assemblerに指示し，**PROC**疑似命令によって作られるプロシージャのデフォルトの型 (nearまたはfar) をコントロールします。**.MODEL**疑似命令によって定義されたメモリモデルは，Turbo Cで使われている同名モデルと互換性があります。

簡略セグメント疑似命令**.CODE, .DATA, .DATA?, .FARDATA, .FARDATA?, .CONST**は，Turbo Cと互換性のあるセグメントを作ります。

次のプログラムはDOTOTAL.ASMという名のTurbo Assemblerモジュールです。

```
        DOSSEG                          ; インテル社方式のセグメントオーダを選択
        .MODEL  SMALL                   ; スモールモデル(nearコードとデータ)
        .DATA                           ; TCコンパチブルの初期化データセグメント
        EXTRN   _Repetitions:WORD       ; 外部宣言
        PUBLIC  _StartingValue          ; 他モジュールから参照される
_StartingValue  DW  0
        .DATA?                          ; TCコンパチブルの未初期化データセグメント
RunningTotal    DW  ?
        .CODE                           ; TCコンパチブルのコードセグメント
        PUBLIC  _DoTotal
_DoTotal        PROC                    ; ファンクション(スモールモデルからnear
                                        ; コール可能)
        mov     cx,[_Repetitions]       ; 繰り返し回数
        mov     ax,[_StartingValue]
        mov     [RunningTotal],ax       ; 初期値をセット
TotalLoop:
        inc     [RunningTotal]          ; RunningTotal + 1
        loop    TotalLoop
        mov     ax,[RunningTotal]       ; 合計値を返す
        ret
_DoTotal        ENDP
        END
```

(*DoTotal*プロシージャ内のラベルの多くに下線 (_) プレフィクスが付いています。これらは通常Turbo Cが必要とするために付けられています。これについての詳細は「下線記号」の節で説明します)。

アセンブラ プロシージャ*_DoTotal*は，スモールモデルのTurbo Cプログラムから次の文でコールすることができます。

```
DoTotal();
```

この例の_*DoTotal*は，プログラムの外部で外部変数*Repetitions*が定義されていることを前提としています。同様に変数*StartingValue*はパブリック変数にされており，プログラムの他の部分からアクセスすることができます。次のTurbo CモジュールSHOWTOT.Cは，DOTOTAL.ASM内のパブリック変数にアクセスし，DOTOTAL.ASMに外部データを引渡しています。

```
extern int StartingValue;
extern int DoTotal(Void);
int Repetitions;
main()
{
    int i;
    Repetitions = 10;
    StartingValue = 2;
    printf("%d¥n",DoTotal());
}
```

SHOWTOT.CとDOTOTAL.ASMから実行可能プログラムSHOWTOT.EXEを作るためには，次のコマンドラインを入力します。

```
tcc showtot dototal.asm
```

_*DoTotal*をコンパクトモデルのCプログラムとリンクする場合は，**.MODEL**疑似命令を**.MODEL** *COMPACT*に変更します。DOTOTAL.ASM内でfarセグメントを使う場合は，**.FARDATA**疑似命令を使います。

Turbo Cプログラムとリンクするために適切なセグメントオーダ，メモリモデル，セグメント名を作ることなどが，簡略セグメント疑似命令の用途です。

従来のセグメント疑似命令とTurbo C

従来のセグメント疑似命令を使ってCプログラムにTurbo Assemblerコードを組み込むのはめんどうなことです。たとえばDOTOTAL.ASM内の簡略セグメント疑似命令を従来のセグメント疑似命令に置き換えると次のようになります。

```
DGROUP group    _DATA,_BSS
_DATA  segment  word public 'DATA'
       EXTRN    _Repetitions:WORD    ; 外部宣言
       PUBLIC   _StartingValue       ; 他モジュールから参照されるラベル
_StartingValue  DW  0
_DATA  ends
_BSS   segment  word public 'BSS'
RunningTotal    DW  ?
_BSS   ends
_TEXT  segment  byte public 'CODE'
       assume   cs:_TEXT,ds:DGROUP,ss:DGROUP
       PUBLIC   _DoTotal
_DoTotal        PROC                 ; 関数(スモールモデルからnearコール可能)
       mov      cx,[_Repetitions]    ; 繰り返し回数
       mov      ax,[_StartingValue]
       mov      [RunningTotal],ax    ; 初期値をセット
TotalLoop:
       inc      [RunningTotal]       ; RunningTotal + 1
       loop     TotalLoop
       mov      ax,[RunningTotal]    ; 合計値を返す
       ret
_DoTotal        ENDP
_TEXT  ENDS
       END
```

従来のセグメント疑似命令を使うとプログラムが長くなるだけでなく,読みにくくなります。また別のCメモリモデルに適合させるための変更を加えることも難しくなります。アセンブラコードをTurbo Cプログラム内に組み込む場合は,従来のセグメント疑似命令を使うとメリットはありません。これらの疑似命令を使ってアセンブラコードをTurbo Cプログラム内に入れたければ,Cプログラムが使うメモリモデルに合ったセグメントを使わなくてはなりません。Turbo Cのセグメント使用法の概要については,Turbo Cのユーザーズガイドを参照してください。

Turbo Cプログラムとのリンクにおいて従来のセグメント疑似命令を使うためには,任意のメモリモデルのTurbo Cプログラムモジュールを-Sオプションでコンパイルするのがもっとも簡単です。これによってTurbo Cはアセンブラバージョンのコードを作ります。このCコード内にはTurbo Cが使った従来のセグメント疑似命令すべてが含まれていますので,これらをアセンブラコード内にコピーしてください。たとえばコマンドライン

```
tcc -s showtot.c
```

を入力すると,ファイルSHOWTOT.ASMが作られます。このファイルの内容を次に示します。

```
        ifndef  ??version
?debug  macro
        endm
```

```
        endif
        name    showtot
_TEXT   segment byte public 'CODE'
DGROUP  group   _DATA,_BSS
        assume  cs:_TEXT,ds:DGROUP,ss:DGROUP
_TEXT   ends
_DATA   segment word public 'DATA'
_d@     label  byte
_d@w    label  word
_DATA   ends
_BSS    segment word public 'BSS'
_b@     label  byte
_b@w    label  word
        ?debug  C E91481D5100973686F77746F742E63
_BSS    ends
_TEXT   segment byte public 'CODE'
;       ?debug  L 3
_main   proc    near
;       ?debug  L 6
        mov     word ptr DGROUP:_Repetitions,10
;       ?debug  L 7
        mov     word ptr DGROUP:_StartingValue,2
;       ?debug  L 8
        call    near ptr _DoTotal
        push    ax
        mov     ax,offset DGROUP:_s@
        push    ax
        call    near ptr _printf
        pop     cx
        pop     cx
@1:
;       ?debug  L 9
        ret
_main   endp
_TEXT   ends
_BSS    segment word public 'BSS'
_Repetitions    label   word
        db      2 dup (?)
;       ?debug  C E9
_BSS    ends
_DATA   segment word public 'DATA'
_s@     label  byte
        db      37
        db      100
        db      10
        db      0
_DATA   ends
        extrn   _StartingValue:word
_TEXT   segment byte public 'CODE'
        extrn   _DoTotal:near
        extrn   _printf:near
_TEXT   ends
        public  _Repetitions
        public  _main
        end
```

_DATA（初期化済みデータセグメント），_TEXT（コードセグメント），_BSS（初期化されてないデータセグメント）のセグメント疑似命令は，疑似命令GROUPおよびASSUMEとともにいつでもアセンブルできる形態になっています。したがってプログラマはこれらをこのまま使うことができます（セグメント疑似命令については第10章で詳しく説明します）。

セグメントデフォルト ： セグメントはいつセットすべきか

Cからコールされるアセンブラ関数は，状況に応じてデータアクセスのためにDSやESに値をセットすることもあります。またアセンブラコードは同等な2つのセグメントレジスタを利用することもあります。したがってTurbo Cからのコール時における複数のセグメントレジスタの設定関係を知っておく必要があります。このセクションでは，Turbo Cからアセンブラ関数を呼び出すときのセグメントレジスタの設定，セグメントレジスタ間の関係を説明しました。またアセンブラ関数はどんな場合に（1つまたは複数の）セグメントレジスタに値をセットしなければならないかについても説明します。

Turbo Cからアセンブラ関数をコールした直後は，CSレジスタとDSレジスタの設定は下表のようになっています。設定は使っているメモリモデルによって異なります（SSは常にスタックセグメントとして使われ，ESは常にスクラッチセグメントレジスタ（作業用）として使われます）。

表6.1 Turbo Cからアセンブラ関数をコールした直後のレジスタの設定

モデル	CS	DS
タイニィ	_TEXT	DGROUP
スモール	_TEXT	DGROUP
コンパクト	_TEXT	DGROUP
ミディアム	*filename*_TEXT	DGROUP
ラージ	*filename*_TEXT	DGROUP
ヒュージ	*filename*_TEXT	*calling_filename*_DATA

filename :アセンブラモジュールの名前
calling_filename： アセンブラモジュールを呼び出したモジュールの名前

タイニィモデルでは，_TEXTとDGROUPは同じです。したがって関数の先頭ではCSとDSは同じです。またタイニィ，スモール，ミディアムの各モデルでは，関数の先頭ではSS

とDSは同じです。

ではCからコールされるアセンブラ関数は，いつセグメントレジスタをセットしたらよいのでしょう？プログラマは直接CSやSSレジスタをセットする必要はありません。CSはfar型のコール，ジャンプ，リターンの際に必要に応じて自動的にセットされ，それ以外では変更されません。SSは常にスタックセグメントを示し，プログラム実行中は変更されることはありません（ただしプログラムで意図的にスタックを変更する場合は別です。スタックを変更した場合は，そのことを十分**意識してください**)。

ESは常に好きなように使うことができます。ESを使ってfarデータをポイントしたり，ESにストリング命令のデスティネーションセグメントをセットしたりすることができます。

ではDSレジスタはどうでしょうか？ヒュージモデル以外のすべてのTurbo Cモジュールでは，関数の先頭ではDSがスタティクデータセグメント（**DGROUP**）を指しますが，通常のこのセグメントはそのままにしておきたい場所です。DSで一時的にfarデータを指してfarデータをアクセスしたいことがありますが，ESを使えば常にfarデータをアクセスすることができます。ESを使えばプログラム内にセグメントオーバライドを使う必要がなくなります。たとえば次のどちらかの方法でfarセグメントにアクセスすることができます。

```
     .
     .
     .
     .FARDATA
Counter DW      0
     .
     .
     .
     .CODE
     PUBLIC  _AsmFunction
_AsmFunction    PROC
     .
     .
     .
     mov     ax,@fardata
     mov     es,ax               ; ESにfarデータセグメントをセット
     inc     es:[Count]          ; カウンタ変数をインクリメント
     .
     .
     .
_AsmFunction    ENDP
     .
     .
     .
```

または

```
      .
      .
      .
      .FARDATA
Counter DW      0
      .
      .
      .
      .CODE
      PUBLIC  _AsmFunction
_AsmFunction    PROC
      .
      .
      .
      assume  ds:@fardata
      mov     ax,@fardata
      mov     ds,ax                  ; DSにfarデータセグメントをセット
      inc     [Count]                ; カウンタ変数をインクリメント
      assume  ds:@data
      mov     ax,@data
      mov     ds,ax                  ; DSをDGROUPに戻す
      .
      .
      .
_AsmFunction    ENDP
      .
      .
      .
```

2番目の例は，farデータセグメントのアクセスのたびにESオーバライドを使わなくてよいというメリットがあります。DSでfarセグメントをポイントした場合は，(2番目の例のように）**DGROUP**内の変数にアクセスする前に必らずDSを復帰してください。たとえアセンブラ関数内で**DGROUP**にアクセスしなくても，関数からリターンする前に必らずDSを復帰してください。なぜならTurbo Cは関数がDSを変更していないことを前提にしているからです。

Cから呼び出されるヒュージモデルの関数では，DSの扱いが少し異なります。ヒュージモデルではTurbo Cは**DGROUP**を使わないかわりに，モジュールごとに固有のデータセグメントを持っています。このようなセグメントは，プログラム内の他のモジュールから見るとfarセグメントとなります。したがってヒュージモデルの関数の先頭では，DSがそのモジュールのfarセグメントを示すようにセットし，関数からリターンするまで変えてはなりません。次に例を示します。

```
        .
        .
        .
        .FARDATA
        .
        .
        .
        .CODE
        PUBLIC  _AsmFunction
_AsmFunction    PROC
        push    ds
        mov     ax,@fardata
        mov     ds,ax               ; DSにfarデータセグメントをセット
        .
        .
        .
        pop     ds
        ret
_AsmFunction    ENDP
        .
        .
        .
```

この例では，関数*AsmFunction*の先頭で**PUSH**命令でDSを退避し，リターン直前に**POP**命令でDSを復帰しています。ヒュージモデルでは，この例のように関数内でDSを退避しなければなりません。

パブリック変数と外部変数

アセンブラプログラムからC関数を呼び出したり，Cの外部変数を参照することがあります。逆にCのプログラムからアセンブラのパブリック変数をコールしたり，アセンブラのパブリック変数を参照することがあります。前のセクションで説明したように，Turbo Cで使えるセグメントがTurbo Assembler内で準備されると，Turbo CとTurbo Assemblerの間で関数と変数を共有するために以下に示す簡単なルールが必要となります。

下線記号

Turbo Cでは，外部ラベルは通常下線記号（_）ではじめる規則があります。Cプログラム内で使われる関数や外部変数には，Turbo Cが自動的にこのプレフィクスが付けます。したがってアセンブラコード内でのみ下線記号に気をつければよいでしょう。アセンブラプログラムから参照するC関数やC変数は，下線記号で始まることに注意してください。またCモジュールから参照するパブリック変数やパブリック関数は，すべて下線記号ではじめなくてはなりません。次にCプログラムの例を示します。

```
extern int ToggleFlag();
int Flag;
main()
{
   ToggleFlag();
}
```

このCプログラムと次のアセンブラプログラムは正常にリンクすることができます。

```
      DOSSEG
      .MODEL    SMALL
      .DATA
      EXTRN    _Flag:word
      .CODE
      PUBLIC   _ToggleFlag
_ToggleFlag      PROC
      cmp      [_Flag],0                 ; フラグ=0か？
      jz       SetFlag                   ; Yes,フラグをセットする
      mov      [_Flag],0                 ; No,フラグをリセットする
      jmp      short EndToggleFlag       ; 終了
SetFlag:
      mov      [_Flag],1                 ; フラグをセットする
EndToggleFlag:
      ret
_ToggleFlag      ENDP
      END
```

*SetFlag*などのCから参照されないローカルなラベルは，その先頭に下線記号を付ける必要はありません。

ところで**-u**コマンドラインオプションを指定すると，下線記号を使わないようにすることができます。このオプションは便利です。しかしTurbo Cのすべてのランタイムライブラリは下線記号を有効にしてコンパイルされています。**-u**オプションを使うためには，ランタイムライブラリのソースコードを購入して，下線記号を無効にして再コンパイルする必要があります（**-p**オプションの詳細は少し後の「Pascalの呼び出し慣例」の節を参照してください。下線記号の使用と大文字小文字の区別を取り止めます）。

大文字と小文字の区別

Turbo Assemblerでは，通常シンボル名を扱う際に大文字小文字の区別をしません。Cでは大文字小文字の区別をするため，少なくともCとアセンブラで共用するラベルに関してはTurbo Assemblerでも大文字小文字の区別をすることをお勧めします。**/MX**と**/ML**を使えば大文字小文字の区別が可能となります。

/MLコマンドラインスイッチは，すべてのシンボルに対して大文字小文字の区別を指定します。一方**/MX**コマンドラインスイッチは，共用変数（**PUBLIC**），外部変数

(**EXTRN**), グローバル変数 (**GLOBAL**), 共通変数 (**COMM**) のラベルに対してのみ大文字小文字の区別を指定します。

ラベルの型

アセンブラプログラムは変数を任意のサイズ (8ビット, 16ビット, 32ビットなど) のデータとして自由にアクセスすることができますが, 一般的に変数はオリジナルサイズでアクセスするのがよいでしょう。たとえば次のようにバイト変数にワードを書き込むとエラーとなります。

```
        .
        .
        .
SmallCount   DB   0
        .
        .
        .
        mov   WORD PTR [SmallCount],0ffffh
        .
        .
        .
```

Turbo AssemblerがC変数に対するアクセスサイズを決める基準は, プログラム内の宣言文しかありません。したがってアセンブラプログラム内で**EXTRN**を使ってCの外部変数を宣言する際に, 変数の適切なサイズを指定することが大切です。たとえばCプログラム内に

```
char c
```

という宣言文がある場合は, 次のアセンブラプログラム例は重大なエラーを発生する可能性があります。

```
        .
        .
        .
EXTRN c:WORD
        .
        .
        .
int   [c]
        .
        .
        .
```

この例では変数cを256回インクリメントするとオーバフローしてしまいます。また変数*c*をワード変数として誤って宣言しているため, **OFFSET** *c*+**1**のバイトは誤ってインクリ

メントされます。そのため予期せぬエラーが発生します。

Cとアセンブラのデータの関係を次に示します。

Cのデータ型	アセンブラのデータ型
unsigned char	byte
char	byte
enum	word
unsigned short	word
short	word
unsigned int	word
int	word
unsigned long	dword
long	dword
float	dword
double	qword
long double	tbyte
near	word
far	dword

far外部変数はセグメントの外に置く

Turbo Assemblerでは，セグメント内で宣言される変数はそのセグメントに関係するシンボルだけです。したがって簡略セグメント疑似命令を使った場合は，farセグメント内シンボルの**EXTRN**宣言はどのセグメント内にも置いてはなりません。これは簡略セグメント疑似命令の欠点です。Turbo Assemblerはセグメント外で**EXTRN**宣言されたシンボルの正当性または妥当性（addressability）をチェックしません。そのため正しいセグメントがセットされていない場合は，必要に応じてセグメントオーバライドを使うこともその変数のアクセス時にその旨を通知することもできません。それでもTurbo Assemblerはそのような外部変数への参照命令を正しくアセンブルします。しかしセグメントの正当性または妥当性チェックは通常どおり行なうことはできません。

（このような欠点があるにもかかわらず）それでもなお簡略セグメント疑似命令を使いたい場合は，従来のセグメント疑似命令を使ってそれぞれの外部変数シンボルがあるセグメントを明示的に宣言し，それぞれのシンボルの**EXTRN**疑似命令をセグメントの先頭に置

いてください。ただし次のような方法もあります。farデータをアクセスするたびに正しいセグメントがセットされていることを確かめるなら，farシンボルの**EXTRN**宣言をセグメントの外に置くことができます。この方法がもっとも簡単です。次にFILE1.ASMとFILE2.ASMをリンクする場合を例にとってみます。それぞれのファイルの内容は次のとおりです。FILE1.ASMの内容は，次のようになっているとします。

```
      .
      .
      .
      .FARDATA
FileVariable   DB   0
      .
      .
      .
```

FILE2.ASMの内容は次のように仮定します。

```
      .
      .
      .
      .DATA
      EXTRN  FileVariable:BYTE
      .CODE
Start   PROC
      mov    ax,SEG FileVariable
      mov    ds,ax
      .
      .
      .
```

この場合**SEG** *FileVariable*は正しいセグメントを返しません。**EXTRN**疑似命令はFILE2.ASMの**DATA**疑似命令の範囲内に置かれています。したがってTurbo Assemblerは，*FileVariable*の存在場所がFILE2.ASMのfarの**DATA**セグメントではなく，FILE2.ASMのnearの**DATA**セグメントであると認識します。

SEG *FileVariable*が正しいセグメントを返すようにするには，FILE2.ASMを次のように変更します。

```
      .
      .
      .DATA
@curseg ENDS
      EXTRN  FileVariable:BYTE
      .CODE
Start   PROC
      mov    ax,SEG FileVariable
      mov    ds,ax
      .
      .
```

この例では，疑似命令@**curseg ENDS**によって**.DATA**セグメントを終了させています。したがって*FileVariable*が**EXTRN**宣言される時点では有効なセグメント疑似命令は1つもありません。

リンカのコマンドライン

Turbo CモジュールとTurbo Assemblerモジュールのもっとも簡単なリンク方法は，Turbo Cのコマンドラインによって Turbo Cにすべての仕事を任せることです。適切なコマンドラインさえ入力すれば，Turbo CはCプログラムをコンパイルした後でTurbo Assemblerを起動してアセンブル処理を実行し，TLINKを起動してオブジェクトファイルをリンクして実行可能ファイルを作ります。たとえばMAIN.CとSTAT.Cという2つのCソースファイルとSUMM.ASMとDISPLAY.ASMという2つのアセンブラファイルからプログラムを作る場合は，次のようにコマンドラインを入力します。

```
tcc main stat summ.asm display.asm
```

この例ではMAIN.CとSTAT.Cがコンパイルされ，SUMM.ASMとDISPLAY.ASMがアセンブルされてそれぞれのオブジェクトファイルが作られます。次にそれら4個のオブジェクトファイルがリンクされ(このときCのスタートアップコードと必要なライブラリ関数もいっしょにリンクされます)，MAIN.EXEという実行可能ファイルができあがります。アセンブラファイル名を指定する際に，.ASM拡張子を付け忘れさえしなければ正しく処理されます。 TLINKをスタンドアロンモードで使う場合は，Turbo Assemblerによって作られたオブジェクトファイルはCのオブジェクトモジュールと同じ標準オブジェクトモジュールとして扱われます。

Turbo AssemblerとTurbo Cの相互作用

Cで使えるアセンブラモジュールの作り方とリンクの方法がわかりました。次にCプログラムから呼び出せるアセンブラ関数にはどのような処理が必要かを説明します。この節で解説する項目は，パラメータの受け渡し，レジスタの使い方(退避・復帰)，呼び出し側プログラムへの値の返しかたの3点です。

パラメータの受け渡し

Turbo Cプログラムはスタック経由でアセンブラ関数にパラメータを引き渡します。関

数を呼び出す前に，Cプログラムはパラメータをスタックにプッシュします。このとき右端のパラメータから順番にプッシュし，左端のパラメータを最後にプッシュします。次に例を示します。

```
.
.
.
Test(i, j, 1)
.
.
.
```

という関数呼び出しは，次のようにコンパイルされます。

```
mov    ax,1
push   ax
push   word ptr DGROUP:_j
push   word ptr DGROUP:_i
call   near ptr _Test
add    sp,6
```

この例では最初に右端のパラメータ1を，次に*j*を，最後に*i*を順番にプッシュしています。

関数からリターンするときもパラメータはスタック上にそのまま残っています。しかしもう用がないので，関数呼び出しの後でスタックポインタは関数呼び出しの前の状態に戻されます。このようにしてパラメータは捨てられます。この例では3個のパラメータのサイズはそれぞれ2バイトなので，合計6バイト分のスタックが使われます。したがって*Test*をコールした後は，スタックポインタに6を加えてパラメータを捨てています。ここでのキーポイントは，Cの関数呼び出し慣例では**呼び出し側**プログラムがスタック内のパラメータを捨てなければならないことです(少し後の「Pascalの呼び出し慣例」を参照してください)。

アセンブラ関数はBPレジスタをベースポインタとして，スタックに入れられたパラメータを受け取ることができます。前の例の関数*Test*の内容を次に示します。

```
     DOSSEG
     .MODEL  SMALL
     .CODE
     PUBLIC  _Test
_Test   PROC
     push    bp
     mov     bp,sp
     mov     ax,[bp+4]          ; パラメータ#1をロード
     add     ax,[bp+6]          ; パラメータ#1とパラメータ#2を加える
     sub     ax,[bp+8]          ; 合計からパラメータ#3を引く
     pop     bp
     ret
_Test   ENDP
     END
```

*Test*はCプログラムからのパラメータをスタック経由で受け取るために，BPレジスタを使うことがわかります(BPレジスタはスタックセグメントをアドレシングすることを忘れないでください)。ではBPを使って**どのように**パラメータをロードするのでしょうか？

図6.4は*Test*の先頭命令を実行する直前のスタックの状態を示します。

```
i = 25;
j = 4;
Test(i, j, 1);
```

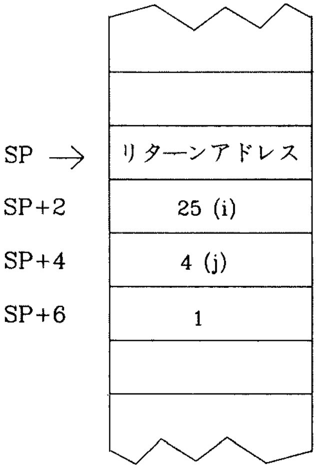


図6.4　Testの先頭命令の実行直前のスタックの状態

Testに渡すパラメータは**SP**で指されるアドレスに入ります。パラメータは，関数呼び出しの直後にリターンアドレスをプッシュしたアドレスより**2**バイト大きいスタックアドレスから入っています。**BP**に**SP**をロードした後は，**BP**を使ってパラメータをアクセスすることができます。ただし呼び出し側の**C**プログラムは**BP**レジスタが変化しないことを前提としているため，関数の先頭で**BP**を退避する必要があります。**BP**をプッシュするとスタックのすべてのオフセットが変化します。図**6.5**はこれらの命令を実行した直後のスタックの状態を示します。

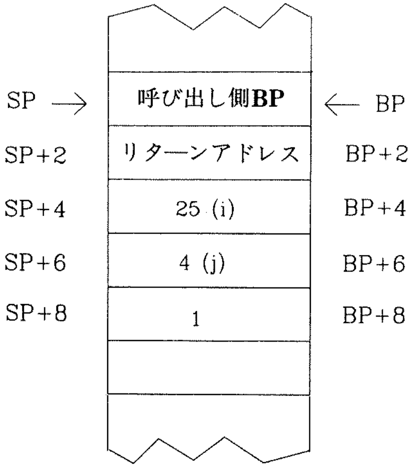


図6.5 PUSHとMOV実行直後のスタックの状態

これはCの標準的なスタックのしくみ，つまりスタックでの関数パラメータと自動変数のしくみです。Cプログラムがいくつパラメータを持っていても左端のパラメータは常にスタックのリターンアドレスの次のスタックアドレスにストアされ，以降のパラメータも順番にスタックにストアされます。渡されたパラメータの順番と型を知っていさえすれば，スタック上の位置はすぐにわかります。

自動変数の領域は，SPの値からあるバイト数を引くことによって確保されます。たとえば100バイトの自動配列のための領域は，*Test*の先頭で次の命令を実行することによって確保することができます（図6.6も参照してください）。

```
.
.
.
push   bp
mov    bp,sp
sub    sp,100
.
.
.
```

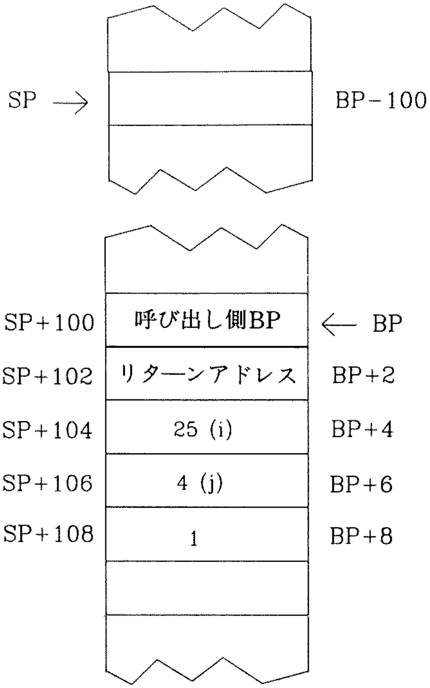


図6.6 PUSH, MOV, SUB実行直後のスタックの状態

自動変数がストアされているスタック領域はBPより下位のアドレスであるため，自動変数をアドレシングするためにはBPに対して負のオフセットを使います。次に例を示します。

```
mov  byte ptr [bp-100],0
```

この例ではさきほど確保した100バイト配列の先頭バイトに0をセットします。一方渡されたパラメータはBPに対して正のオフセットを指定してアドレシングされます。

さきに示したとおり，必要なら自動変数に対する領域を確保することができます。Turbo Assemblerには，自動変数の領域確保と命名を簡単に行なうための特殊な疑似命令**LOCAL**があります。プロシージャ内に**LOCAL**疑似命令があると，そのプロシージャに対して自動変数を定義します。次に例を示します。

```
LOCAL    LocalArray:BYTE:100,LocalCount:WORD=AUTO_SIZE
```

この例では自動変数*LocalArray*と*LocalCount*を定義します。*LocalArray*は実際には[BP-100]に割り当てられたイコール定義で，*LocalCount*は[*BP-102*]に割り当てられたイコール定義です。しかしこれらのラベルを変数名として使えば，値がわからなくても変数を

指定することができます。*AUTO_SIZE*は必要とする自動メモリのトータルバイト数です。自動変数の領域を確保するためには，SPからこの値を引かなくてはなりません。

次に**LOCAL**の使用例を示します。

```
     .
     .
_TestSub  PROC
     LOCAL      LocalArray:BYTE:100,LocalCount:WORD=AUTO_SIZE
     push bp                          ; 呼び出し側プログラムのスタック退避
     mov  bp,sp                       ; スタックポインタをロード
     sub  sp,AUTO_SIZE                ; 自動変数用に領域を確保
     mov  [LocalCount],10             ; ローカルカウント変数に10をセット
                                      ; LocalCount = [BP-102]
     .
     .
     .
     mov  cx,{LocalCount]             ; ローカル変数からカウントをロード
     mov  al,'A'                      ; 'A'をストアする
     lea  bx,[LocalArray]             ; ローカル配列をポイント
FillLoop:
     mov  [bx],al                     ; 次のバイトをストア
     inc  bx                          ; 次のバイトをポイント
     loop FillLoop                    ; 次のバイトがあれば処理続行
     mov  sp,bp                       ; 自動変数用の領域を開放
                                      ; add sp,AUTO_SIZEでも可
     pop  bp                          ; 呼び出し側プログラムのスタックを復帰
     ret
_TestSub  ENDP
     .
     .
```

この例では自動変数の定義行（LOCAL）の第1フィールドは，変数のデータ型（**BYTE, WORD, DWORD, NEAR**など）であることに注意してください。第2フィールドはこの変数型に対して確保する要素数です。このフィールドはオプション（任意指定項目）です。指定した場合は自動配列を定義し，省略した場合は指定した型の1要素を確保します。したがって*LocalArray*は100バイトの要素から構成され，*LocalCount*は1ワードの要素から構成されます。

この例の**LOCAL**行は＝*AUTO_SIZE*で終わっています。=で始まるこのフィールドはオプションです。指定した場合は=以降のラベルに必要な自動変数用メモリのバイト数を割り当てます。**LOCAL**疑似命令はラベルを定義するだけで，命令コードを生成したりデータ領域を確保するものではありません。したがって自動変数用のメモリの確保と開放のためには，以降このラベルを使わなくてはなりません。つまり**LOCAL**は自動変数を確保する疑似命令ではなく，ラベルを定義するだけの疑似命令です。以後このラベルを使って自動変数のメモリ確保やアクセスを行なうことができます。

LOCALのたいへん便利な機能は，自動変数のラベルとその合計サイズのラベルの有効範

囲を，使われているプロシージャの範囲内に限定することです。したがって同じ名前の自動変数を他のプロシージャ内に自由に使うことができます。

これまで説明したように**LOCAL**を使えば，自動変数を簡単に定義したり使用したりすることができます。ただし第10章でも説明したとおり，**LOCAL**をマクロ命令内で使うとまったく意味が異なります(**LOCAL**疑似命令のこれらの用途については，**リファレンスガイド**の第3章を参照してください)。

Turbo Cはここで説明したようにスタックを扱います。いくつかのTurbo Cモジュールを**-S**オプションを付けてコンパイルして得られるアセンブラコードを見れば，Turbo Cがどのようにスタック領域を確保して使っているかがわかります。

ここまでは理解できました。しかしさらに憶えなくてはならないことがあります。常にBPの固定オフセットでパラメータをアクセスすることはめんどうです。これは間違いのもとになるばかりでなく，パラメータを追加すると関数の他のパラメータのスタック構成が変ってしまいます。たとえば*Test*を4つのパラメータを受け取る関数に変更してみます。

```
Test(Flag, i, j, 1);
```

この場合，パラメータ*i*はオフセット4からオフセット6に，パラメータ*j*はオフセット6からオフセット8に，という具合にパラメータの位置がずれます。次のようにパラメータオフセットに対してイコール定義を使うことができます。

```
          .
          .
          .
Flag            EQU   4
AddParm1        EQU   6
AddParm2        EQU   8
SubParm1        EQU   10
          mov   ax,[bp+AddParm1]
          add   ax,[bp+AddParm2]
          sub   ax,[bp+SubParm1]
          .
          .
          .
```

これでもオフセットの計算と保守はめんどうです。しかしもっと重要なことがあります。それは，farコードモデルではプッシュされるリターンアドレスは，nearコードモデルに比べて2バイトだけ大きくなることです。farコードモデルのコードポインタやfarデータモデルのデータポインタのサイズも同様に2バイト大きくなります。したがってメモリモデルの種類を問わず，スタックを正常にアクセスすることができる関数を作ることは困難に思われます。

しかし安心してください。Turbo Assemblerには**ARG**という疑似命令があります。この

疑似命令を使えばアセンブラ関数内でのパラメータの扱いが容易になります。

ARG疑似命令の機能は，指定した変数に対して適切なスタックオフセットを自動的に確保することです。次に例を示します。

```
arg   FillArray:WORD,Count:WORD,FillValue:BYTE
```

この例では，*FillArray*（ワードサイズパラメータ），*Count*（ワードサイズパラメータ），*FillValue*（バイトサイズパラメータ）の3つのパラメータをアクセスします。ARG疑似命令はラベル*FillArray*に［BP+4］を，ラベル*Count*に［BP+6］を，ラベル*FillValue*に［BP+8］をそれぞれ割り当てています。このARGは非常に重宝なものです。なぜなら，ARGで定義したラベルは割り当てられた値がわからなくても使うことができます。

例として，Cからコールされる*FillSub*という名の次のような関数について考えてみましょう。

```
main()
{
#define ARRAY_LENGTH 100
    char TestArray[ARRAY_LENGTH];

    FillSub(TestArray,ARRAY_LENGTH,'*');
}
```

*FillSub*内にARG疑似命令を使ってパラメータを処理します。

```
_FillSub  PROC NEAR
     ARG    FillArray:WORD,Count:WORD,FillValue:BYTE
     push   bp                 ; 呼び出し側のスタックフレームを退避
     mov    bp,sp              ; SPをロード
     mov    bx,[FillArray]     ; ストア先の配列へのポインタをロード
     mov    cx,[Count]         ; ワード長をロード
     mov    al,[FillValue]     ; ストアする値をロード
FillLoop:
     mov    [bx],al            ; 1文字ストア
     inc    bx                 ; 次の文字をポイント
     loop   FillLoop           ; 次の処理へ
     pop    bp                 ; 呼び出し側のスタックフレームを復帰
     ret
_FillSub  ENDP
```

渡されたパラメータをARGを使って扱うと，こんなに手間が省けます。ARGはnearリターンとfarリターンによるサイズの違いを自動的に調節します。さらにARGで定義したラベルの有効範囲はそれらが使われているプロシージャ内に限定されます。したがって別のプロシージャで同じパラメータ名を使うことができます。

ARG疑似命令の詳細は，**リファレンスガイド**の第3章を参照してください。

レジスタの退避

Turbo Cに関する限り，Cからコールされるアセンブラ関数はBP, SP, CS, DS, SSレジスタを退避しさえすれば，どんな処理でも行なうことができます。これらのレジスタは関数の処理実行中に変更できますが，関数からリターンするときに呼び出し前と同じ状態に戻す必要があります。AX, BX, CX, DX, ESおよびフラグは任意に変更することができます。

Turbo CはSIとDIはレジスタ変数として特別に扱います。アセンブラ関数を呼び出すCモジュール内でレジスタ変数が使われている場合は，SIとDIを退避しなければなりません。ただしレジスタ変数が使われていない場合は，その必要はありません。レジスタ変数が使われているか否かに関わらず，SIとDIは常に退避する習慣をつけておくのがよいでしょう。アセンブラモジュールとCモジュールをリンクする場合，またはレジスタ変数を有効にしてCコードを再コンパイルする場合は，（レジスタを退避するために）必らずアセンブラコードを変更する必要があることを覚えておいてください。

値の返し方

Cから呼び出されるアセンブラ関数は，C関数と同様に値を返すことができます。関数は値を次のように返します。

返す値の型	値のストア先
unsigned char	AX
char	AX
enum	AX
unsigned short	AX
short	AX
unsigned int	AX
int	AX
unsigned long	DX AX
long	DX AX
float	8087 スタックトップ(TOS)レジスタ(ST(0))
double	8087 スタックトップ(TOS)レジスタ(ST(0))
long double	8087 スタックトップ(TOS)レジスタ(ST(0))
near	AX
far	DX AX

通常，関数は8ビットと16ビットの値はAXレジスタに返し，32ビットの値はDX：AXに返します（上位16ビットがDXに入ります）。また浮動小数点の値はST（0）（8087のスタックトップ（TOS）レジスタに返します。浮動小数点エミュレータを使っている場合は8087エミュレータのTOSレジスタに返します。

構造体はもう少し複雑です。1バイト長または2バイト長の構造体はAXに返し，4バイト長の場合はDX：AXに返します。3バイト長と5バイト長以上の構造体は静的データエリアにストアし，その静的データのポインタを返します。すべてのポインタと同様，構造体のnearポインタはAXに返し，farポインタはDX：AXに返します。

ではスモールモデルのCから呼び出されるアセンブラ関数，*FindLastChar*を例にとってみましょう。この関数は与えられた文字列の最終文字のポインタを返します。この関数を呼び出すCの文は次のようになります。

```
extern char * FindLastChar(char * StringToScan);
```

ここで*StringToScan*が文字列の名前で，この文字列の最終文字のポインタを返します。*FindLastChar*関数の内容を次に示します。

```
        DOSSEG
        .MODEL  SMALL
        .CODE
        PUBLIC _FindLastChar
_FindLastChar   PROC
        push    bp
        mov     bp,sp
        cld                     ; ストリング命令の設定(カウントアップ)
        mov     ax,ds
        mov     es,ax           ; ESにnearデータセグメントのポインタをセット
        mov     di,             ; ES:DIで文字列先頭を指す
        mov     al,0            ; 文字列終端のヌルをサーチ
        mov     cx,0ffffh       ; 最大64K-1バイトをサーチ
        repnz   scasb           ; ヌルが見つかるまでサーチ
        dec     di              ; ヌルまでポインタを戻す
        dec     di              ; 最終文字までポインタを戻す
        mov     ax,di           ; nearポインタをAXにストア
        pop     bp
        ret
_FindLastChar    ENDP
        END
```

この関数は，文字列の最終文字のポインタ（near）を求めてAXレジスタに返しています。

Turbo CからTurbo Assembler関数を呼び出す

次にTurbo Assembler関数を呼び出すTurbo Cのプログラムについて考えてみましょう。下のTurbo AssemblerモジュールCOUNT.ASM内には，*LineCount*という関数があります。この関数は，与えられた文字列内の行数と文字数を返します。

```
; スモールモデルのCからコールされるアセンブラ関数
; 機能: 0を終端とする文字列の行数と文字数をカウントする
;
; 関数プロトタイプ:
;   extern unsigned int LineCount(char * near StringToCount,
;                                 unsigned int near * CharacterCountPtr);
; 入力:
;    char near * StringToCount: 行数をカウントする文字列へのポインタ
;    unsigned int near * CharacterCountPtr:
;                               文字数をストアするint変数へのポインタ
;
NEWLINE  EQU     0ah                ; 改行コード(Cのnewline)
         DOSSEG
         .MODEL  SMALL
         .CODE
         PUBLIC  _LineCount
_LineCount       PROC
         push    bp
         mov     bp,sp
```

```
        push    si                  ; 呼び出し側プログラムのレジスタ変数が
                                    ; あれば退避
        mov     si,[bp+4]           ; SIに文字列アドレスセット
        sub     cx,cx               ; 文字カウンタに0をセット
        mov     dx,cx               ; 行カウンタに0をセット
LineCountLoop:
        lodsb                       ; 次の文字をロード
        and     al,al               ; ヌル(文字列終端)か?
        jz      EndLineCount        ; Yes...終わり
        inc     cx                  ; No....文字カウンタ+1
        cmp     al,NEWLINE          ; newlineか?
        jnz     LineCountLoop       ; No....次の文字をチェック
        inc     dx                  ; Yes...行カウンタ+1
        jmp     LineCountLoop
EndLineCount:
        inc     dx                  ; ヌル文字で終わる行をカウント
        mov     bx,[bp+6]           ; 文字カウントのストア先をポイント
        mov     [bx],cx             ; 文字カウントをセット
        mov     ax,dx               ; 行カウントを関数の値としてセット
        pop     si                  ; 呼び出し側プログラムのレジスタ変数が
                                    ; あれば復帰
        pop     bp
        ret
_LineCount      ENDP
        END
```

次のCモジュールCALLCT.Cで*LineCount*関数を呼び出しています。

```
char * TestString="Line 1¥nline 2¥nline 3";
extern unsigned int LineCount(char * StringToCount,
       unsigned int * CharacterCountPtr);
main()
{
     unsigned int LCount;
     unsigned int CCount;

     LCount = LineCount(TestString, &CCount);
     printf("Lines: %d¥nCharacters: %d¥n", LCount, CCount);
}
```

これらの2つのモジュールは次のコマンドラインでコンパイルされ，リンクされます。

```
tcc -ms callct count.asm
```

この例からもわかるとおり，スタックフレームのポインタサイズとロケーションは他のモジュール内では変るため，*LineCount*はスモールモデルのCプログラムとリンクしてはじめて機能します。次のモジュールはラージモデルのCプログラムから呼び出される，(*LineCount*の別のバージョン) COUNTLG.ASMです (farポインタを受け取らず，*LineCount*がfar宣言してあれば，スモールモデルではありません)。

```
; ラージモデルのCからコールされるアセンブラ関数
; 機能: 0を終端とする文字列の行数と文字数をカウントする
;
; 関数プロトタイプ:
;   extern unsigned int LineCount(char * far StringToCount,
;          unsigned int far * CharacterCountPtr);
; 入力:
;   char far * StringToCount: 行数をカウントする文字列へのポインタ
;   unsigned int far * CharacterCountPtr:
;                               文字数をストアするint変数へのポインタ
;
NEWLINE  EQU     0ah                  ; 改行コード(Cのnewline)
         DOSSEG
         .MODEL  LARGE
         .CODE
         PUBLIC  _LineCount
_LineCount       PROC
         push    bp
         mov     bp,sp
         push    si                   ; 呼び出し側プログラムのレジスタ変数が
                                      ; あれば退避
         push    ds                   ; Cの標準データセグメントを退避
         lds     si,[bp+6]            ; SIに文字列アドレスセット
         sub     cx,cx                ; 文字カウンタに0をセット
         mov     dx,cx                ; 行カウンタに0をセット
LineCountLoop:
         lodsb                        ; 次の文字をロード
         and     al,al                ; ヌル(文字列終端)か?
         jz      EndLineCount         ; Yes...終わり
         inc     cx                   ; No....文字カウンタ+1
         cmp     al,NEWLINE           ; newlineか?
         jnz     LineCountLoop        ; No....次の文字をチェック
         inc     dx                   ; Yes...行カウンタ+1
         jmp     LineCountLoop
EndLineCount:
         inc     dx                   ; ヌル文字で終わる行をカウント
         les     bx,[bp+10]           ; 文字カウントのストア先をポイント
         mov     es:[bx],cx           ; 文字カウントをセット
         mov     ax,dx                ; 行カウントを関数の値としてセット
         pop     ds                   ; Cの標準データセグメントを復帰
         pop     si                   ; 呼び出し側プログラムのレジスタ変数が
                                      ; あれば復帰
         pop     bp
         ret
_LineCount       ENDP
         END
```

このCOUNTLG.ASMとCALLCT.Cは次のコマンドラインでリンクされます。

```
tcc -ml callct countlg.asm
```

Pascalの呼び出し慣例

ここまでは,Cプログラムがどのようにアセンブラ関数へパラメータを引き渡すのかを説明しました。Cプログラムは，右側のパラメータから順番にプッシュし，関数を呼び出し，関数呼び出し後にスタックからパラメータを捨てます。Turbo CはまたPascalで使われている慣例にしたがって関数を呼び出すことができます。この慣例では，左側のパラメータから順番にプッシュし，**呼び出された**プログラムがスタックからパラメータを捨てます。Turbo CでこのPascalの慣例を利用したい場合は, **-p**コマンドラインオプションまたは**pascal**キーワードを指定します。

Pascal慣例を使うアセンブラ関数の例を次に示します。

```
;
; コール形式: TEST(i, j, k);
;
i       equ     8                       ; 一番左のパラメータ
j       equ     6
k       equ     4                       ; 一番右のパラメータ
;
        DOSSEG
        .MODEL  SMALL
        .CODE
        PUBLIC  TEST
TEST    PROC
        push    bp
        mov     bp,sp
        mov     ax,[bp+i]               ; iをロード
        add     ax,[bp+j]               ; i+j
        sub     ax,[bp+k]               ; i+j-k
        pop     bp
        ret     6                       ; リターン(パラメータの6バイトを捨てる)
TEST    ENDP
        END
```

図6.7は命令**MOV BP, SP**を実行した直後のスタックの状態を示します。

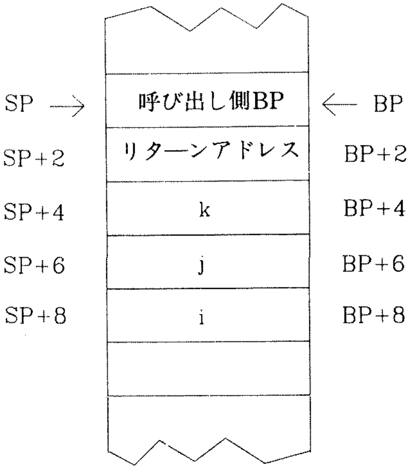


図6.7　MOV BP, SP実行直後のスタックの状態

RET 6は，呼び出された関数がスタックからパラメータを捨てるために実行します。

Pascalの呼び出し慣例では，外部シンボルとパブリックシンボルはすべて大文字でなければなりません。また先頭に下線記号は付けません。ところで，なぜCプログラム内でPascalの呼び出し慣例を利用するのでしょうか？　Pascal慣例を利用すると，通常のCプログラムより若干コンパクトで高速になります。これは関数呼び出し後に**ADD SP** ***n***の命令を実行する必要がないためです。Pascalの呼び出し慣例（コーリングシーケンス）の詳細については，第7章を参照してください。

Turbo AssemblerからTurbo Cを呼び出す

特殊な処理を実行するためにCからアセンブラ関数を呼び出すことはよくあることですが，逆にアセンブラプログラムからC関数を呼び出したい場合もあります。Turbo Assembler関数からTurbo C関数を呼び出す際は，アセンブラ側でスタックフレームの処理を行なう必要がありません。そのためCからアセンブラを呼び出す場合より簡単です。アセンブラプログラムからTurbo C関数を呼び出すための必要条件を次に説明します。

Cスタートアップコードのリンク

一般的なルールとして，Cスタートアップモジュールを最初にリンクしたプログラム内のアセンブラモジュールからTurbo Cのライブラリ関数を呼び出すのはよい考えです。この安全な方法には，TC.EXEまたはTCC.EXEコマンドラインでリンクしたプログラムやTLINKで直接リンクしたプログラム（C0T, C0S, C0C, C0M, C0L, C0Hのどれかのファイルを最初にリンク）が含まれます。

Cスタートアップモジュールをリンクしていないプログラムからは，Turbo Cライブラリ関数を呼び出してはなりません。そのようなプログラムから呼び出すと，Turbo Cライブラリ関数の中には正常に動作しないものもあるからです。どうしても呼び出す必要がある場合は，スタートアップモジュールのソース（Turbo Cの提供ディスクのC0.ASMというファイル）をチェックして，Cライブラリのソースを購入することをお勧めします。そうすれば必要なライブラリ関数を適切に初期化することができます。別の方法として，ライブラリ関数を抜粋してアセンブラプログラムとリンクすることもできます。たとえば，アセンブラプログラムX.ASM（機能はライブラリ関数を呼び出すだけ）とライブラリ関数cm.libをリンクする場合は，次のコマンドラインを入力します。

```
tlink x,x,,cm.lib
```

ここで*m*はメモリモデルの先頭文字（タイニィモデルなら*t*，スモールモデルなら*s*など）を表わします。TLINKが未定義シンボルを通知した場合は，プログラムにCスタートアップモジュールをリンクしないとそのライブラリ関数を呼び出すことはできません。

注意：ユーザ定義のC関数からCライブラリ関数を呼び出すことは，ライブラリ関数を直接呼び出すのと同じです。Cスタートアップモジュールがないと，Cライブラリ関数を直接・

間接に呼び出すアセンブラプログラムにトラブルが発生することがあります。

セグメントの設定が正しいことを確認する

すでに説明したように，Turbo C関数とTurbo Assemblerモジュールが同じメモリモデルを使い，両者が使うセグメントが一致していることが必要条件です。メモリモデルとセグメントを一致させる方法については，少し前の「Turbo CとTurbo Assemblerのインターフェースの概要」の節を参照してください。また全セグメントの外側または必要なセグメントの内側に必らず**EXTRN**疑似命令を入れてください。

呼び出しの実行

Turbo Cの関数呼び出しを実行するために必要な条件は，すでに「Turbo CからTurbo Assembler関数を呼び出す」の節で説明しました。ここでもう一度，C関数呼び出しのメカニズムを簡単に復習してみましょう。ただし，ここではTurbo AssemblerモジュールからTurbo C関数を呼び出すことに重点を置きます。

Turbo C関数へパラメータを引き渡すには，一番右側のパラメータを最初に，一番左のパラメータを最後に順番にスタックにプッシュします。そして関数呼び出しを実行します。たとえばTurbo Cのプログラムで, Turbo Cライブラリ関数**strcpy**を呼び出して*SourceString*を*DestString*に転送するには，次の命令文を実行します。

```
strcpy(DestString, SourceString);
```

同じ処理をアセンブラ関数で行なうと，次のようになります。

```
lea   ax,SourceString      ; 一番右のパラメータ
push  ax
lea   ax,DestString        ; 一番左のパラメータ
push  ax
call  _strcpy              ; 文字列転送の関数呼び出し
add   sp,4                 ; パラメータを捨てる
```

この例では, 呼び出し後でSPを調整してパラメータを捨てていることに注意してください。

Pascalの呼び出し慣例を使ったC関数を呼び出す場合は，一番左のパラメータを最初に，一番右のパラメータを最後にプッシュします。呼び出し後にSPを調整することはしません。

```
lea   ax,DestString       ; 一番左のパラメータ
push  ax
lea   ax,SourceString     ; 一番右のパラメータ
push  ax
call  _strcpy             ; 文字列転送の関数呼び出し
```

いうまでもありませんが，この例ではstrcpyを-pスイッチで再コンパイルしていることを前提にしています。**strcpy**の標準ライブラリバージョンは，Pascalの呼び出し慣例ではなくCの呼び出し慣例に従うからです。「値の返しかた」の節でも説明したとおり，C関数は8ビットと16ビットの値はAXレジスタに返し，32ビットの値はDX：AXに返します（上位16ビットがDXに入ります）。浮動小数点の値は8087のTOSレジスタに返します。また構造体はそのサイズによってさまざまな方法で返します。

C関数はSI, DI, BP, DS, SS, SPの各レジスタを退避しますが，AX, BX, CX, DX, ESおよびフラグは任意に変更します。

Turbo Assemblerモジュールから Turbo C関数を呼び出す

Turbo AssemblerモジュールからTurbo C関数を呼び出したいケースとして，複雑な計算を伴う場合があります。計算式はアセンブラよりもCで記述した方が簡単だからです。整数や浮動小数点数が混在する計算では特にその必要があります。このような計算はアセンブラだけで行なうこともできますが，型変換や浮動小数点の計算はCで行なうのが簡単です。

Turbo C関数を呼び出すアセンブラ関数の例を下に示します。このアセンブラプログラムは浮動小数点の計算を実行します。この例では，Turbo CプログラムがTurbo Assembler関数に複数の整数を引き渡します。アセンブラ関数はその合計を計算した後に別のTurbo C関数を呼び出して，それらの平均値を浮動小数点付きで計算します。

CALCAVG.CのCコードの部分は次のとおりです。

```
extern float Average(int far * ValuePtr, int NumberOfValues);
#define NUMBER_OF_TEST_VALUES 10
int TestValues[NUMBER_OF_TEST_VALUES] = {
   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};

main()
{
    printf("The average value is: %f¥n",
           Average(TestValues, NUMBER_OF_TEST_VALUES));
}

float IntDivide(int Dividend, int Divisor)
{
    return( (float) Dividend / (float) Divisor );
}
```

AVERAGE.ASMのアセンブラコードの部分は次のとおりです。

```
;
; Turbo Cからコールされ，整数の平均値を返すスモールモデルの関数
; 最後の除算ではC関数IntDivide()をコールする
;
; 関数プロトタイプ
;       extern float Average(int far * ValuePtr, int NumberOfValues);
;
;入力：
;     int far * ValuePtr:             ; 平均値の配列
;     int NumberOfValues:             ; 平均をとる値の数
;
        DOSSEG
        .MODEL  SMALL
        EXTRN   _IntDivide:PROC
        .CODE
        PUBLIC  _Average
_Average        PROC
        push    bp
        mov     bp,sp
        les     bx,[bp+4]               ; ES:BXで数値配列をポイント
        mov     cx,[bp+8]               ; 平均をとる値の数
        mov     ax,0                    ; 合計値クリア
AverageLoop:
        add     ax,es:[bx]              ; 現在値を加える
        add     bx,2                    ; 次の値をポイント
        loop    AverageLoop
        push    WORD PTR [bp+8]         ; 値の数を右端パラメータとして
                                        ; IntDivideに渡す
        push    ax                      ; 合計値を左端パラメータとして渡す
        call    _IntDivide              ; 浮動小数点付きで平均値を計算
        add     sp,4                    ; パラメータを捨てる
        pop     bp
        ret                             ; 平均値は8087のTOSレジスタにストア
_Average        ENDP
        END
```

Cの**メイン**プログラムは，整数配列*TestValues*へのポインタと配列の長さをアセンブラ関数*Average*に渡します。*Average*は整数を合計し，合計値と値の数をC関数*IntDivide*に渡します。*IntDivide*は合計値と値の数を浮動小数点数にキャスト（型変換）し，平均値を計算します(アセンブラでやれば数ステップかかる計算を1行で実行しています)。*IntDivide*は8087のTOSレジスタ経由で*Average*に平均値を返し，*Average*はそのまま**main**に平均値を返しています。

CALCAVG.CとAVERAGE.ASMのコンパイルとリンクを実行して，実行可能プログラムCALCAVG.EXEを作るには，次のコマンドラインを入力します。

```
tcc calcavg.asm
```

この例では, farポインタがすべてのモデルに引き渡されているため, *Average*はプログラムを変更することなくスモールデータモデルとラージデータモデルの両方を扱っています。

ラージコードモデル(ヒュージ, ラージ, ミディアム)をサポートするには, 適切な**.MODEL**疑似命令を使う必要があります。

第7章

Turbo Assemblerと Turbo Pascalのインターフェース

Turbo Assemblerには，アセンブリ言語のコードをTurbo Pascalのプログラムに組み込むことができる強力な機能があります。この章では，多数の例題プログラムを用いて，これらの機能をフルに利用するために知っていなければならないことをすべて説明します。ほとんどのプログラムは，プログラムの全体をTurbo Pascalで書くことができます。またTurbo Pascalでは，他のほとんどのPascalと異なり，*Port*[], *PortW* [], *Mem*[], *MemW* [], *MemL* [] ，などの配列を使ってマシンのすべての資源に直接アクセスすることができます。また*Intr*手続き（プロシージャ）や*MsDos*手続きを使ってBIOSやオペレーティングシステムをコールすることができます。

それでは何故，アセンブラのプログラムをPascalのプログラムと組み合わせて使用するのでしょうか？　主な理由は2つあります。Turbo Pascalから直接に行なうことができない比較的少数の処理を行なうためと，直接アセンブラ命令で書くことによりサイズが小さくて高速のルーチンが書ける利点を利用するためです（Turbo Pascalのコンパイラ自体はアセンブリ言語で書かれていて，非常に高速です)。この章では，Turbo Pascalのプログラムに，アセンブリ言語の強力な威力を組み込む方法を説明します。

注意：Turbo Pascal は，バージョン番号を特に指定しない限り，バージョン4.0以降のバージョンです。

Turbo Pascalのメモリマップ

Turbo Pascalのプログラムに組み込むアセンブリ言語のコードを書きはじめる前に，Pascalのコンパイラが，メモリの中に情報をどのように配置するかを理解することが大切です。Turbo Pascalのメモリモデルは，第4章で説明したミディアムモデルとラージモデルの両方の特徴を兼ね備えています。グローバル変数と型つき定数は，単一のグローバルデータセグメントに格納され，DSレジスタからのオフセットで高速にアクセスできます。各ユニットは別々のコードセグメントに格納されます。またヒープ領域は，利用可能なすべての領域に伸びることができます。Turbo Pascalでは，メモリのどこにある対象にもアクセスできるように，アドレスは通常far（32ビット）ポインタで受け渡しされます。

Turbo Pascalのメモリマップは次のようになります。

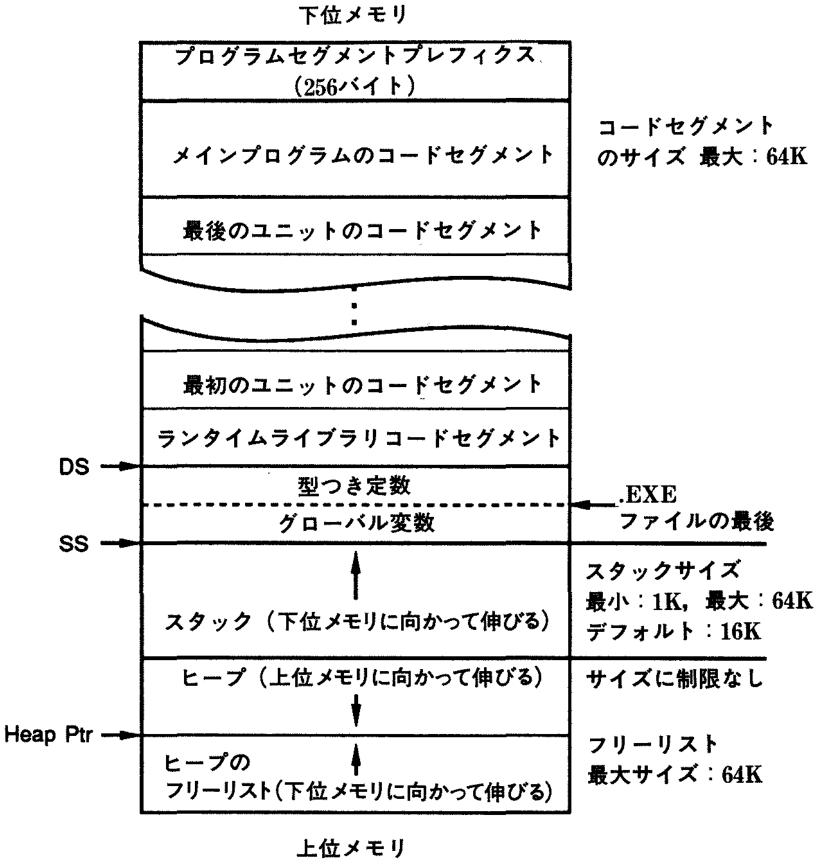


図7.1 Turbo Pascal 5.0のプログラムのメモリマップ

プログラムセグメントプレフィクス

プログラムセグメントプレフィクス（PSP）は，プログラムがロードされるときにMS-DOSによって作成される256バイトの領域です。この領域には，そのプログラムを呼び出すときに用いられるコマンドラインパラメータ，使用可能なメモリのサイズ，***DOSの環境変数***を指すポインタ（DOSが使う文字変数のリスト）などが格納されます。

Turbo Pascal 3.0では，PSPのセグメントアドレスは，コード全体のセグメントアドレスと同じでした。しかし4.0以降のバージョンでは，PSP，メインプログラム，メインプログラムが使用する各ユニット，およびランタイムライブラリコードは，別々のセグメントに格納されます。PSPのセグメントアドレスは，あらかじめ宣言されたグローバル変数***PrefixSeg***にセットされます。この***PrefixSeg***の値をベースアドレスとして使用することにより，PSPセグメント内の情報にアクセスすることができます。

コードセグメント

Turbo Pascalのプログラムは，少なくとも2つのコードセグメントを持ちます。1つはメインプログラムのセグメント，もう1つはランタイムライブラリコードのセグメントです。さらに各ユニットは，別々のコードセグメントに格納されます。各セグメントのサイズは最大64Kbです。64Kbを越えるプログラムを生成するために従来オーバーレイを使っていたプログラマは，バージョン4.0以降では，より高速に実行するためにすべてのコードをメモリに常駐させることができます。Turbo Assemblerのルーチンは，コードセグメント（セグメント名が**CODE**または**CSEG**）に組み込まれます。

グローバルデータセグメント

Turbo Pascalのグローバルデータセグメントは，ランタイムライブラリコードセグメントの次（上位側）に配置されます。グローバルデータセグメントには，初期化されたデータ（**型つき定数**）と初期化されないデータ（**グローバル変数**）が64Kbまで格納されます。Turbo Pascal 3.0と同じように，型つき定数は実際には定数ではなく，プログラムがロードされるときあらかじめ初期化されている変数です。Turbo Pascal 3.0と異なりTurbo Pascal 4.0では，型つき定数は，コードセグメントではなくグローバルデータセグメントに入れら

れます。その結果，型つき定数へのアクセスが，Turbo Pascal 3.0よりも高速になります。Turbo Assemblerのモジュールから参照するときは，グローバルデータセグメントの名前は**DATA**または**DSEG**です。

スタック

Turbo Pascalのバージョン4.0またはそれ以降のバージョンでは，スタックはグローバルデータセグメントの上位側に配置されます。この配置はTurbo Pascal 3.0の配置と異なることに注意してください。スタックとヒープは，互いに相手に向かって伸びるのではなく，反対側に伸びます（スタックは下位メモリに向かって，ヒープは上位メモリに向かって伸びます)。そのためスタックには，ある決まった大きさのメモリ領域が割り当てられます。スタック領域のサイズは，1K（スタックを多く浪費しないプログラムのとき）から64K（再帰呼び出しの回数の多いプログラムのとき）の範囲で選択することができます。デフォルトのサイズは16Kです。16Kのサイズは，ほとんどのプログラムに対して十分以上の大きさです。スタックとヒープのサイズは，コンパイラ指令$ *M*で指定することできます。

ほとんどの80x86用のプログラムと同じようにTurbo Pascalのプログラムでも,スタックポインタは最初スタックセグメントの最上位を指し,スタックにデータが積まれるにしたがって下位メモリを指します。Turbo Pascalは通常,手続きや関数が呼び出されたとき,スタックがスタックセグメントを越えるかどうかチェックするコードを生成します。{$*S*-}指令を指定すると,このチェックコードの生成を抑制することができます。

ヒープ

Turbo Pascalのプログラムのメモリマップの最上位はヒープです。デフォルトでは,コードセグメント,データセグメント,スタックセグメントに使用されないすべてのメモリがヒープになります。ただし$*M*指令により，最大サイズを制限することもできます（また*$M*指令により，少なくともある大きさのヒープ領域が確保できないときは，プログラムが動作しないようにすることもできます)。

ヒープ領域は,最下位のアドレスから上位メモリに向って,手続き*New*や*GetMem*により，動的変数に割り当てられます。割り当てられた領域は，手続き（プロシージャ）*Dispose, Release, FreeMem*により解放することができます。Turbo Pascal 4.0では，*Dispose*や*FreeMem*を使用して領域を解放すると，ヒープ内の空き領域のアドレスとサイズが，**フリーリスト**という名前のデータ構造を使って記録されます。このフリーリストは，サイズが64Kまで拡大す

ることができ，ヒープ領域の最上位のアドレスから下位メモリに向って拡大します。

Turbo Pascalのレジスタの使い方

Turbo Pascal 4.0でも，Turbo Pascal 3.0と同じように，レジスタの使用には最小限の制約しかありません。関数や手続きを呼び出すときは，スタックセグメント(SS)，データセグメント（DS)，ベースポインタ（BP）の3つのレジスタの値を保存するだけで済みます。DSはグローバルデータセグメント（**DATA**と呼ばれる）のセグメントアドレスを指し，SSはスタックセグメントを指します。BPは，関数や手続きにより，**アクチベーションレコード**（手続きや関数が，呼び出しパラメータ，ローカル変数，一時的格納領域を指すポインタなどを格納するために使用するスタック領域）を参照するために使用されます。すべてのサブプログラムは，リターンする前に，パラメータがスタックに残らないようにスタックポインタ（SP）を調節しなければなりません。

nearかfarか

Turbo Pascalのプログラムは複数のコードセグメントを含むので，関数や手続きを呼び出すためにnearコールとfarコールの両方が使用されます。nearコールとfarコールの違いは何でしょうか？　nearコールは，同じコードセグメント内にある手続きや関数しか呼び出せません。それに対してfarコールは，メモリのどこにある手続きや関数でも呼び出すことができます。そのかわりfarコールは，nearコールよりも時間がかかり，メモリスペースも余計に必要です。

Turbo Pascalのプログラムのサブプログラムは，この2つの呼び出しのどちらかで呼び出されるように書かれていなければなりません。Turbo Pascalのコンパイラは，サブプログラムの宣言に基づいて自動的に正しいコールタイプを選択してくれます。ユニットの**インターフェース**部に宣言されているサブプログラムは，他のセグメントからfarコールで呼び出すことができるように，いつでもfar型になります。これに対してメインプログラムの中で宣言されているか，またはユニットの**実現部**でのみ宣言されているサブプログラムは，near型になります（ただし｛$ *F*+｝指令を指定してコンパイルすると，強制的にfar型にすることができます）。

Turbo Pascalのプログラムにリンクするアセンブラルーチンを書くときは，そのルーチンのコールタイプが，Pascalのプログラムからの呼び出しのタイプと合っていることを確かめなければなりません。Pascalのプログラムにおける**external**宣言からはfar型でなければならないアセンブラルーチンをnearと宣言しても，Turbo Pascalはエラーを報告しません。

Pascalのプログラムとアセンブラルーチンの情報の共用

$Lコンパイラ指令とexternalサブプログラム

Turbo AssemblerのルーチンをTurbo Pascalのプログラムに組み込むための2つの手段は，{$*L*} コンパイラ指令と**external**サブプログラム宣言です。指令 {$L MYFILE.OBJ} は，Turbo Pascalのコンパイラに，DOSの**標準のリンカブル（リンク可能）**オブジェクト形式のファイルMYFILE.OBJを探し，Turbo Pascalのプログラムにリンクするように指示します。$Lの後のファイル名に拡張子を指定しないときは .OBJであるとみなされます。

Turbo Assemblerの手続きや関数をTurbo Pascalのプログラムの内部から見えるようにするには，それらをPUBLICシンボルとして宣言し，そしてそのPascalプログラムの中で**external**宣言しなければなりません。

Turbo Pascalのプログラムの中の**external**宣言は，**forward**宣言と非常によく似ています。

```
procedure AsmProc(a : integer; b : real); external;

function AsmFunc(c : word; d : byte); external;
```

これらの宣言は，Turbo Assemblerのプログラムの中の次のような宣言に対応します。

```
CODE      SEGMENT BYTE PUBLIC
AsmProc   PROC NEAR
          PUBLIC AsmProc
          .
          .
          .
AsmProc   ENDP
AsmFunc   PROC FAR
          PUBLIC AsmFunc
          .
          .
          .
AsmFunc   ENDP
CODE      ENDS
```

Turbo Pascalの**external**サブプログラム宣言は，プログラムやユニットの最も外側のレベルになければなりません。これ以外のレベルで**external**宣言すると，コンパイル時にエ

ラーになります。

Turbo Pascalのコンパイラは，アセンブラルーチンの**PROC**疑似命令のnearまたはfarオペランドが，Pascalのプログラムからの呼び出しの種類（nearコールまたはfarコール）に対応しているかどうかチェックしません。実際，パブリックラベル*AsmProc*と*AsmFunc*が，**PROC**の名前であるかどうかさえチェックしません。アセンブラルーチンのコールタイプとPascalの宣言（呼び出しのタイプが決まる）を整合させることはプログラマの責任です。

PUBLIC疑似命令：アセンブラルーチンの情報をPascalプログラムが利用できるようにする

Turbo Pascalのプログラムには，アセンブラルーチンで**PUBLIC**宣言されたラベルだけが見えます。アセンブラルーチンからPascalのプログラムに参照させられるのはラベルだけです。アセンブラルーチンで**PUBLIC**宣言されたラベルには，Turbo Pascalのプログラム中に対応する**external**サブプログラム宣言がなければなりません。もしなければ，コンパイラはエラーを報告します。パブリックラベルは，**PROC**宣言の一部である必要はありません。Turbo Pascalに関する限り

```
AsmLabel    PROC FAR
            PUBLIC AsmLable
```

と

```
AsmLabel:
            PUBLIC Asmlabel
```

は同等です。

EXTRN疑似命令：Pascalプログラムの情報をアセンブラルーチンが利用できるようにする

Turbo Assemblerのモジュールは，リンクされるTurbo Pascalのプログラムやユニットの最も外側のレベルで宣言された手続き，関数，変数，型つき定数にアクセスすることができます（そのアセンブラモジュールに対する{$*L*}指令や**external**宣言の後で宣言された変数にもアクセスできることに注意してください）。Turbo Pascalのラベルや通常の定数は，アセンブラルーチンからは見えません。

Turbo Pascalのプログラムにおいて，次のようなグローバル変数が宣言されているもの

とします。

```
var
    a : byte;
    b : word;
    c : shortInt;
    d : integer;
    e : real;
    f : single;
    g : double;
    h : extended;
    i : comp;
    j : pointer;
```

次のようにEXTRN宣言することにより，アセンブラルーチンからアクセスすることができます。

```
EXTRN A : BYTE    ; 1バイト
EXTRN B : WORD    ; 2バイト
EXTRN C : BYTE    ; アセンブリ言語では符号付きも符号なしも同様に扱う
EXTRN D : WORD    ; 同上
EXTRN E : FWORD   ; 6バイトのソフトウェア実数
EXTRN F : DWORD   ; 4バイトIEEE浮動小数点数
EXTRN G : QWORD   ; 8バイトIEEE倍精度浮動小数点数
EXTRN H : TBYTE   ; 10バイトIEEE浮動小数点数
EXTRN I : QWORD   ; 8087  8バイト符号付き整数
EXTRN J : DWORD   ; Turbo Pascalポインタ
```

同様にして，Turbo Pascalの手続きや関数（ライブラリルーチンを含む）にもアクセスすることができます。たとえば，次のようなTurbo Pascalのユニットがあるとします。

```
unit Sample;
{ アセンブラの手続きから呼ばれるいくつかのPascalの手続きを定義する例題ユニット }

interface
 procedure TestSample;      { 外部から見えるのでfarでなければならない }
 procedure PublicProc;      { 外部から見えるのでfarでなければならない }

implementation
var
  A : word;
procedure AsmProc; external;
{$L ASMPROC.OBJ}
procedure PublicProc;
  begin   { PublicProc }
    Writeln('In PublicProc');
  end;    { PublicProc }

procedure NearProc;          { nearでなければならない }
  begin   { NearProc }
    Writeln('In NearProc');
  end;    { NearProc }
```

```
{$F+}
procedure FarProc;           { コンパイラ指令より強制的にfar }
  begin    { FarProc }
    Writeln('In FarProc');
  end;     { FarProc }

{$F-}
procedure TestSample;
  begin    { TestSample }
    Writeln('In TestSample');
    A := 10;
    Writeln('Value of A before ASMPROC = ',A);
    AsmProc;
    Writeln('Value of A after ASMPROC = ',A);
  end;     { TestSample }
end.
```

手続き*AsmProc*は，次のように**EXTRN**疑似命令を使って外部宣言することにより，***PublicProc, NearProc, FarProc***を呼び出すことができます。

```
DATA      SEGMENT WORD PUBLIC
          ASSUME DS:DATA
          EXTRN A:WOERD                 ; ユニットからの変数
DATA      ENDS

CODE      SEGMENT BYTE PUBLIC
          ASSUME CS:CODE
          EXTRN  PublicProc : FAR       ; far手続き(ユニットによりエクスポートされる)
          EXTRN  NearProc   : NEAR      ; near手続き(ユニットに対してローカル)
          EXTRN  FarProc    : FAR       ; far手続き(ローカルだが強制的にfar)

AsmProc   PROC NEAR
          PUBLIC AsmProc
          CALL   FAR PTR PublicProc
          CALL   NearProc
          CALL   FAR PTR FarProc
          mov    cx,ds:A                 ; ユニットから変数Aを取り込む
          sub    cx,2                    ; 変更を加える(2を引く)
          mov    ds:A,cx                 ; 変数Aを元の場所に格納
          RET
AsmProc   ENDP
CODE      ENDS
          END
```

次のメインプログラムにより，このPascalのユニットとアセンブラのコードをテストすることができます。

```
program TSample;
uses Sample;
begin
  TestSample;
end.
```

コマンドライン版のTurbo AssemblerとTurbo Pascalを使って，このPascalのプログラムとアセンブラのルーチンから例題プログラムを作成して実行するには，次のバッチファイルコマンドを使用します。

```
TASM ASMPROC
TPC /B TSAMPLE
TSAMPLE
```

外部サブプログラムは，Turbo Pascalのプログラムの最も外側のレベルで宣言しなければならないので，外部サブプログラムから**EXTRN**宣言をしてPascalプログラムの手続きや関数のローカルなシンボルにアクセスすることはできません。ただしPascalのプログラムからアセンブラのサブプログラムを呼び出すときに，値パラメータまたは変数パラメータとして渡すことができます。

EXTRN宣言を使用するときの制約

Turbo Pascalでは，同じ識別名が複数のユニットで使用されている場合に，その識別名の前にユニット名とピリオドをつけて(**修飾つき識別名**)，どのユニットの識別名であるかを指定することができます。この規則は，Turbo Assemblerではサポートされないのでエラーになります。したがって

```
EXTRN SYSTEM.Assign : FAR
```

のように宣言すると，Turbo Assemblerはエラー メッセージを表示します。

Turbo Pascalの識別名を**EXTRN**宣言して使用する上で，さらに2つの小さな制約があります。1つは，EXTRN宣言した手続きや関数の識別名をアドレスの演算に使えないことです。たとえば

```
EXTRN PublicProc : FAR
```

のように宣言したときは，

```
call PublicProc + 42
```

のような文を書くことはできません。もう1つの制約は，Turbo Pascalのリンカは，ワードをバイトに分割する演算子は認識できないため，**EXTRN**宣言したラベルにこれらの演算子を適用することはできない（バイト単位のアクセスはできない）ということです。たとえば

```
EXTRN i :WORD
```

と宣言したら，Turbo Assemblerのモジュールの中では，**LOW** *i*や**HIGH** *i*のような式は使用できません。

セグメント参照の解決の利用

Turbo Pascalは，システムのメモリ内のどのアドレスにでもロードすることができる.EXEファイルを生成します。そのためプログラムのセグメントがどこにロードされるかあらかじめわからないため，リンカはDOSの.EXEローダに，プログラムをロードするときにプログラム中のすべてのセグメント参照を解決するように指示します。セグメント参照が解決された後は，すべてのセグメント参照は正しい値になります。

Turbo Assemblerの**SEG**演算子を使うと，この機能を利用して実行時に，指定したセグメントのセグメントアドレスに置き換えることができます。たとえばプログラムの中でDSの値を変更する必要があるが，元の値をスタック上に退避させたり，一時的に他の場所に移動するためにムダなCPU時間を浪費したくないとしましょう。そのような場合には，次のような行を挿入することにより，DSに元の値を再ロードすることができます。

```
.
.
mov ax, SEG DATA   ; Turbo PascalのDSの実際のセグメントアドレスを取得する
mov ds,ax          ; セグメントアドレスをDSにロードする
.
.
```

DOSは，Turbo Pascalのプログラムがロードされるときに，**MOV**命令の即値オペランドフィールドに，**SEG DATA**に対する正しい値を入れます。これはセグメントレジスタに元の値を再ロードする最も速い方法です。

このテクニックはたとえば，割り込みサービスルーチンにより，Pascalプログラムのデータセグメント内のデータにアクセスしたり，データを格納したりする場合にも必要です。割り込みが発生した時点では，DSレジスタにPascalプログラムのデータセグメントの値が入っているとは限りませんが，上のようなアセンブラコードを使うと，データセグメントのアドレスをセットすることができます。

デッドコードの削除

Turbo Pascalには，**デッドコード削除**機能があります。この機能は，最終的な.EXEファイルを書き込むときに，実行されないルーチンのコードを削除します。ただしTurbo Assembler

のモジュールについては完全な情報がないので，限られた最適化しか行ないません。

Turbo Pascalは，.OBJモジュール内の外から見える手続きや関数がまったく呼び出されない場合に限り，そのモジュールのコードを削除します。その反対にそのモジュールの中のどれかのルーチンが参照される場合には，そのモジュールの全体が残ります。

Turbo Pascalのデッドコード削除機能を最も効果的に利用するには，アセンブリプログラムを少数のルーチンしか含まない小さなモジュールに分割するのが得策です。そうすることにより，最終プログラムから無用なコード（もしあれば）を取り除くことができます。

Turbo Pascalのパラメータの引き渡し方法

Turbo Pascalは，スタック（バージョン4.0ではsingle型，double型，extended型，comp型の値パラメータは，数値演算コプロセッサのスタック）を使ってパラメータを渡します。パラメータはいつでも，サブプログラムの宣言の中に現われる順に左から右に評価され，スタックに積まれます。このセクションでは，Turbo Pascalのプログラムからアセンブラルーチンにパラメータを渡す方法について説明します。

値パラメータ

値パラメータは，パラメータが渡されたサブプログラムにおいて，その値を変更することができないパラメータです。多くのコンパイラと異なりTurbo Pascalは，すべての値パラメータをそのままスタックにコピーすることはしません。この後に説明するように，データ型により異なる方法で渡します。

スカラ型

すべてのスカラ型（boolean型，char型，shortint型，byte型，integer型，word型，longint型，部分範囲型および列挙型）の値パラメータは，スタックに積んで渡されます。渡される値パラメータのサイズが1バイトなら，16ビットのワードとしてプッシュされます。ただしそのワードの上位バイトは無意味です（このバイトは，Turbo Pascal 3.0およびそれ以前のバージョンでは0でしたが，それ以降のバージョンでは0とはかぎりません)。値パラメータが2バイトなら，そのままプッシュされます。4バイト長（longint）なら，2つの16ビットのワードとしてプッシュされます。8086ファミリーのプロセッサにおける標準的なやり方にしたがって，上位のワードが先にプッシュされ，スタック上の上位アドレスを占めます。

comp型は整数型ですが，パラメータの受け渡しにおいてはスカラ型と見なされないことに注意してください。この型の値パラメータは，Turbo Pascal 4.0では，CPUのスタックではなく8087のスタックに積んで渡されます。Turbo Pascal 5.0では，CPUのスタックに積んで渡されます。

実数

実数型の値パラメータ（Turbo Pascalの6バイトのソフトウェア浮動小数点型）は，スタック上の6バイトとして渡されます。スタックに積んで渡される値パラメータのうちで4バイトより大きいのは，この型だけです。

Single, Double, Extended, Comp 8087型

Turbo Pascal 4.0では，8087型の値パラメータは，CPUのスタックではなくコプロセッサのスタックに積んで渡されます。8087のスタックは8レベルの深さしかありませんので，Turbo Pascal 4.0のサブプログラムは，8087型の値パラメータは，8つまでしか受け取ることができません。8087型のパラメータは，サブプログラムがリターンする前に，数値演算コプロセッサのスタックからすべてポップしなければなりません。

Turbo Pascal 5.0は，Turbo Cと同じパラメータ受け渡し方法を使って8087型の値パラメータを渡します。すなわち他の型のパラメータと一緒にメインCPUのスタックに積んで渡します。

ポインタ

ポインタ型の値パラメータはすべて，farポインタとして直接スタックにプッシュされます（最初にセグメントアドレスを含むワードが，次にオフセットを含むワードがプッシュされます）。Intelの規則にしたがって，セグメントアドレスが上位側を占めます。Turbo Assemblerのプログラムは，**LDS**命令や**LES**命令を使ってポインタパラメータを取り出すことができます。

文字列

文字列は，長さによらず，通常はスタックにプッシュされません。そのかわりにTurbo Pascalは，その文字列を指すfarポインタをプッシュします。ポインタで指し示される元の文字列を変更したくないときは，サブプログラムでその文字列のコピーを作って，それを変更しなければなりません。

この規則のただ1つの例外は，あるオーバーレイユニット*A*のルーチンが別のオーバーレイユニット*B*のルーチンに文字列定数を値パラメータとして渡す場合です。ここでオーバーレイユニットとは，{$*O*+}を指定してコンパイルしたユニットのことです。この場合に

は，ユニット*A*は呼び出しを行なう前にスタック上に一時格納領域を確保してそこに文字列定数を格納し，ユニット*B*のルーチンにそのスタックアドレスを渡します。詳しい説明はTurbo Pascal 5.0のリファレンスガイド第13章を参照してください。

レコードと配列

長さが1バイト，2バイトまたは4バイトのレコードや配列は，値パラメータとして渡されるときは，スタックに直接コピーされます。レコードや配列がこれ以外のサイズ(3バイトも含む）のときは，ポインタがプッシュされます。1バイト，2バイト，または4バイト長でないレコードや配列をサブプログラムで変更する場合には，必要に応じて，そのレコードや配列のコピーを作らなければなりません。

集合

集合は，文字列と同様に，通常はスタックにプッシュされません。かわりにその集合を指すポインタがプッシュされます。サブプログラムが受け取るポインタは，その集合の〝正規化された″32バイトの表現を指し示します。この集合の最下位のバイトの第1ビットは，序数の0またはそれに対応する基底型（またはその親型）の要素に対応します。

この規則の唯一の例外は，あるオーバーレイユニットAのルーチンが別のオーバーレイユニットBのルーチンに，集合を値パラメータとして渡す場合です。ここでオーバーレイユニットとは，{$*O*+}を指定してコンパイルしたユニットのことです。この場合には，ユニットAは呼び出しを行なう前にスタック上に一時格納領域を確保してそこに集合定数を格納し，ユニットBのルーチンにそのスタックアドレスを渡します。詳しい説明は，Turbo Pascal 5.0のリファレンスガイド第13章を参照してください。

変数パラメータ

変数パラメータはすべて，メモリ内の変数のアドレスを指し示すfarポイタとして渡されます。

スタックの管理

Turbo Pascalは，サブプログラムがリターンする前にスタック上のすべてのパラメータ

を放棄することを要求します。

スタックを調整する方法は2つあります。1つは**RET n**命令（*n*はプッシュしたパラメータのバイト数）を使う方法です。もう1つはレジスタ（またはメモリ）にリターンアドレスを退避して，パラメータを1つずつポップする方法です。8086や8088(ファミリの中で最も遅いプロセッサ）で使用するプログラムを速度に関して最適化するときは，ポップする手法が役に立ちます。なぜならこれらのプロセッサでは，ベースにオフセットを加算するアドレシングは，1回のアクセスに少なくとも8サイクル必要だからです。また**POP**命令は1バイトしか必要としないので，メモリスペースを省略することができます。

注意：**.MODEL, PROC, ARG** 疑似命令を使用しているときは，アセンブラはすべての**RET**命令に対して自動的に，ポップすべきパラメータバイトのバイト数を加えます。

パラメータへのアクセス

Turbo Assemblerのルーチンに制御が移ったときは，スタックの一番上にはリターンアドレス（そのルーチンがnearなら2ワード，farなら4ワード）が入っています。そしてその上位側には，渡されるパラメータが入っています。

注意：スタックポインタからのパラメータの位置を計算するときは，そのルーチンでプッシュしたレジスタ（もしあれば）のバイト数を計算に入れることを忘れないでください。

Turbo PascalのプログラムからTurbo Assemblerのルーチンに渡されるパラメータにアクセスするための3つの基本的な方法があります。

■BPレジスタを使ってスタック内のアドレスを指定する。
■他のレジスタを使ってスタック内のアドレスを指定する。
■リターンアドレスをポップしてから，パラメータをポップする。

1番目と2番目の方法は3番目の方法に比べて複雑なので，次の2つのセクションで説明します。3番目の方法は，リターンアドレスを安全な場所にポップしてから，パラメータをレジスタにポップする方法です。この方法は，アセンブラルーチンが，スタック上にローカル変数のための領域を確保する必要がないときに最適です。

BPレジスタを使ってスタック内のアドレスを指定する方法

Turbo PascalのプログラムからTurbo Assemblerのルーチンに渡されるパラメータにアクセスするための第1の(そして最もよく使用される)方法は，次のようにBPレジスタを使ってスタックのアドレスにアクセスする方法です。

```
CODE       SEGMENT
           ASSUME CS:CODE
MyProc     PROC FAR                       ; 手続きMyProc(i,j : integer);
           PUBLIC MyProc
j          EQU WORD PTR [bp+6]            ; jは退避されたBPとリターンアドレスの
                                          ; 直上 (上位側)
i          EQU WORD PTR [bp+8]            ; iはjの直上
           push bp                        ; 呼出し側のBPを退避
           mov  bp,sp                     ; BPがスタックのトップを指すようにする
           mov  ax,i                      ; パラメータiのBPからの相対アドレスを
                                          ; axにロード
           .
           .
           .
```

この方法でアクセスしようとするパラメータのスタックオフセットを計算するときは，退避されたBPレジスタの2バイトを計算に入れることを忘れないでください。

この例の中の，パラメータのアドレスに対してイコール定義された**同値テキスト**の使い方に注意してください。このようにするとプログラムのコードは読みやすくなりますが，1つ小さな問題が生じます。それは，この種のイコール定義には**EQU**疑似命令しか使用できない（=疑似命令は使用できない）ので，同一のTurbo Assemblerのソースファイルの中では，シンボル*i*や*j*を再定義できないことです。この問題を回避する1つの方法は，パラメータ名が重複しないように，もっと記述的な（内容をよく表わす）長いパラメータ名を使うことです。もう1つの方法は，各ルーチンを別々にアセンブルすることです。

ARG疑似命令

Turbo Assemblerでは，BPレジスタからの相対アドレスを使ってパラメータにアクセスするときは，スタックオフセットを計算してそのアドレスの等値テキストを定義するかわりにもう1つの方法を使用することができます。それは**ARG**疑似命令を使うことです。**ARG**疑似命令は自動的に，BPに対するパラメータのオフセットを決定します。また**ARG**疑似命令は，**RET**命令の中で指定するパラメータブロックのサイズも計算します。**ARG**疑似命令によって生成されるシンボルは，**PROC**で囲まれた範囲内でだけ定義されるので，異なる手続きや関数のパラメータ名は重複していてもかまいません。

前の例を**ARG**疑似命令を使って書き換えると，次のようになります。

```
CODE     SEGMENT
         ASSUME CS:CODE
MyProc   PROC FAR                   ; 手続きMyProc(i,j : integer); external;
         PUBLIC MyProc
         ARG j : WORD, i : WORD = RetBytes

         push bp                    ; 呼び出し側のBPを退避
         mov  bp,sp                 ; BPがスタックのトップを指すようにする
         mov  ax,i                  ; パラメータiのBPからの相対アドレスを
                                    ; axにロード
         .
         .
         .
```

Turbo Assemblerの**ARG**疑似命令は，パラメータに対するローカルシンボルiとjを生成します。上の例の行

```
ARG j : WORD, i : WORD = RetBytes
```

は，シンボル*i*を［**WORD PTR BP+6**］に，シンボル*j*を［**WORD PTR BP+8**］に，シンボル ***RetBytes***を数4（パラメータブロックのバイトサイズ）にイコール定義されます。スタックアドレスは，リターンアドレスのサイズとプッシュされたBPの両方が考慮されています。もし*MyProc*が**NEAR PROC**であれば，*i*は［WORD PTR *BP*＋*4*］に，*j*は［WORD PTR *BP*＋*6*］に，***RetBytes***は4にイコール定義されます（そのためどちらの場合にも，***Myproc***は**RET** ***RetBytes***命令で終わることができます）。

ARG疑似命令を使うときは，パラメータを逆の順番に並べることを忘れないでください。**ARG**疑似命令の中では，手続き（または関数）のヘッダ中の**最後の**パラメータを先頭に，**先頭の**パラメータを最後に置かなければなりません。

Turbo Pascalのプログラムとともに**ARG**疑似命令を使うときは，注意しなければならないことがもう1つあります。他のいくつかの言語と異なり，Turbo Pascalはいつでも，バイトサイズの値パラメータを16ビットのワードとしてプッシュします。そしてTurbo Assemblerのルーチンにおいて，この余計なバイトを考慮するのはユーザの責任です。たとえば，Pascalのプログラム内で次のように宣言されている関数を，アセンブリ言語で書いたとします。

```
function MyProc(i,j : char) : string; external;
```

この関数のための**ARG**疑似命令は，次のようになります。

```
ARG j : BYTE : 2, i :BYTE : 2 = RetBytes RETURNS result : DWORD
```

各引数の後の：2は，Turbo Assemblerに，各文字が2バイトの配列としてプッシュされ

ていることを知らせます（各バイト対の上位バイトは無意味です）。

文字列を返す関数では，**ARG**疑似命令の**RETURNS**オプションにより，変数を，関数結果の一時的な格納場所が入っているスタックアドレスにイコール定義することができます（これについては，すぐ後に説明します）。**ARG**疑似命令の**RETURNS**部の変数は，スタックのパラメータブロック（パラメータの入っている領域）のサイズには影響を及ぼしません。**ARG**疑似命令の詳しい説明は，**リファァレンスガイドの**第3章を参照してください。

.MOEDLとTurbo Pascal

.MOEDL疑似命令で*TPASCAL*を指定すると，簡易セグメント配置，メモリモデル，言語サポート機能をセットアップすることができます。先にアセンブラのプログラムにPascalの手続きや関数としての機能を持たせるための構成を説明しました。同じ例題プログラムを**.MODEL**疑似命令と**PROC**疑似命令を使って書き直すと，次のようになります。

```
        .MODEL TPASCAL
        .CODE
MyProc  PROC FAR i:BYTE,j:BYTE RETURNS result:DWORD
        PUBLIC MyProc
        mov  ax,i
        .
        .
        .
        ret
```

パラメータの順番を逆にしなくてもよいこと，他の多くの文が不要になることに注意してください。.MODEL疑似命令のオペランドに*TPASCAL*を指定すると，Turbo Assemblerは自動的に，Turbo Pascalの呼び出し方法にしたがって，セグメント名の定義，ベースポインタの設定（**PUSH BP**と**MOV BP, SP**），リターン処理（**POP BP**と**RET** *N*）を行ないます。

他のレジスタを使ってスタック内のアドレスを指定する方法

パラメータにアクセスする2番目の方法は，ベースレジスタ（BX）やインデックスレジスタ（SIまたはDI）を使って，スタックからパラメータを取得する方法です。ただし，これらのレジスタのデフォルトのセグメントは，SSではなくてDSであることを忘れないでください。したがってこれらのレジスタを使うには，セグメントオーバーライドプレフィックスによりスタックセグメントを指定するか，セグメントレジスタの値を変更しなければなりません。

次にBXを使ってパラメータを取得する方法を示します。

```
CODE      SEGMENT
          ASSUME CS:CODE
MyProc    PROC FAR                  ; 手続きMyProc(i,j : integer);
          PUBLIC MyProc
j         EQU WORD PTR SS:[BX+4]    ; jはリターンアドレスの上(上位アドレス側)
i         EQU WORD PTR SS:[BX+6]    ; iはjの上
          mov  bx,sp                ; BXがスタックのトップを指すようにする
          mov  ax,i                 ; パラメータiのオフセットをaxにロード
          .
          .
          .
```

パラメータを参照する回数が数が少ないルーチンでは，この方法を使うと時間とメモリを節約できます。なぜならBXを使うと，BPを使用する場合と異なり，ルーチンの最後で退避した値を回復する必要がないからです。

Turbo Pascalの関数の結果

Turbo Pascalの関数は，結果のデータ型により異なる方法で結果を返します。

スカラー型の結果

スカラー型の結果は，CPUのレジスタに返します。1バイトの値はALに，2バイトの値はAXに，4バイトの値はDX：AX（DXに上位ワードが入る）で返します。

実数型の結果

Turbo Pascalの6バイトのソフトウェア実数（real）型の結果は，CPUの3つのレジスタに返します。上位ワードがDXに，中位ワードがBXに，下位ワードがAXに入ります。

8087型の結果

8087型の結果は，8087コプロセッサのスタックトップレジスタST(0)（またはただSTとも表わす）に返します。

文字列型の結果

文字列型の結果は，呼び出しの前にTurbo Pascalにより確保された一時的格納領域に返します。この領域を指すfarポインタは，最初のパラメータをプッシュする前に，スタックにプッシュされます。このポインタは，パラメータリストの一部ではないことに注意してください。

注意：このポインタは，スタックから取り除かないでください。Turbo Pascalでは，関数がリターンした後スタックに，このポインタが残されていなければならないからです。

ポインタ型の結果

ポインタ型の結果はDX：AX（セグメント：オフセット）で返します。

ローカル変数の格納領域の確保

Turbo Assemblerのルーチンは，それ自体の変数——**静的変数**（呼び出しと呼び出しの間も消滅しない変数）と**揮発変数**（呼び出し後に消滅する変数）——のための領域を確保することができます。次の2つのセクションで，その方法を説明します。

静的格納領域の確保

Turbo Assemblerのプログラムは，Pascalのプログラムのグローバルデータセグメント（**DATA**または**DSEG**）内に静的変数の格納領域を確保することができます。この領域を確保するためには，DB, DWのような疑似命令を使って，次のように指定します。

```
DATA     SEGMENT PUBLIC
MyInt    DW ?              ; ワードの確保
MyBtye   DB ?              ; バイトの確保
         .
         .
         .
DATA     ENDS
```

Turbo Assemblerによりグローバルデータセグメントの中に格納された変数には，2つの大きな制約があります。第1に，これらの変数は“プライベート（専用）”すなわちTurbo Pascalのプログラムから見えないことです（ただしこれらの変数を指すポインタを渡すことはできます）。第2に，これらの変数は，型つき定数のようにあらかじめ初期化することができないことです。次のような文

```
MyInt   DW   42              ; このように書いてもMyIntは42に初期化されない
```

は，アセンブラのモジュールをTurbo Pascalのプログラムにリンクするときにエラーにはなりません。ただし，プログラムの実行時に*MyInt*の初期値は42になりません。

Turbo Pascalのプログラムの中で変数または型つき定数を宣言し，Turbo Assemblerのルーチンの中で**EXTRN**疑似命令を使用してそれらの変数が見えるようにすることにより，これらの制約を回避することができます。

揮発格納領域の確保

Turbo Assemblerのルーチンは，呼び出し中に，スタック上に揮発格納領域（ローカル変数）を確保することができます。アセンブラルーチンがリターンするときは，この格納領域を解放し，BPレジスタを回復しなければなりません。次の例の手続き（プロシージャ）*MyProc*は，2つの整数変数*a*と*b*の領域を確保します。

```
CODE    SEGMENT
        ASSUME CS:CODE
MyProc  PROC FAR                              ; 手続きMyProc(i : Integer);
        PUBLIC MyProc
        LOCAL a : WORD, b : WORD = LocalSpace ; [bp-2]をaに，[bp-4]をbに
                                              ; 割り当てる
i       equ word ptr [bp+6]                   ; パラメータiはBPの上(上位アドレス)
        push bp                               ; 呼び出し側のBPを退避
        mov  bp,sp                            ; BPがスタックのトップを指すよう
                                              ; にする
        sub  sp,LocalSpace                    ; 2ワード分の領域を作る
        mov  ax,42                            ; aの初期値をAXレジスタにロード
        mov  a,ax                             ; aを[bp-2]に格納
        xor  ax,ax                            ; AXをクリア
        mov  b,ax                             ; Bを0に初期化
        .                                     ; 必要な処理を行う
        .
        .
        mov  sp,bp                            ; SPを元の値に戻す
        pop  bp                               ; BPを元の値に戻す
        ret  2                                ; ワードパラメータをポップする
MyProc  ENDP
CODE    ENDS
        END
```

この例の**LOCAL**疑似命令は，ローカル変数のシンボルを生成し，生成したシンボルに領域を確保します。次の文

```
LOCAL a : WORD, b : WORD = LocalSpace
```

は，この手続きの呼び出し中，シンボル*a*を［*bp-2*］に，シンボルbを［*bp-4*］に，シンボル*LocalSpace*を4（ローカル変数領域のサイズ）に等値します。パラメータを表わすシンボルを生成するこのような疑似命令はありませんので，EQU疑似命令を使ってイコール定義しなければなりません。

ローカル変数の格納領域を確保し初期化するためのさらにうまい方法は，SPをデクリメントするかわりに，初期値をプッシュすることです。この方法を使うには，**sub sp,** *LocalSpace*を次のように置き換えます。

```
mov   ax,42        ; aの初期値をaxにセット
push ax            ; それをaに格納
xor   ax,ax        ; axをゼロにする
push ax            ; ゼロをbに転送
```

この方法を使うときは，スタックのどこに何が入っているか見失わないようにしなければなりません。シンボルaとbは，プッシュする前に参照しないでください。

これ以外にも，**PUSH CONST**命令を使ってローカル変数を初期化したり（80186, 80286, 80386で使用可能）や，BPをスタックにプッシュするかわりにレジスタに退避する（使用しないレジスタがあれば）ことにより最適化することができます。

Turbo Pascalのプログラムとリンクするアセンブリ言語ルーチンの例

この節では，Turbo Pascalから呼び出すことができるアセンブラルーチンの例を示します。

汎用の16進数への変換ルーチン

次のルーチンは，*num*の指し示す場所にある数（バイト列）を，長さが*byteCount*＊2の16進数の数字の列に変換します。各バイトは2つの数字（2バイト）に変換されますので，*byteCount*の最大値は127です(チェックしません)。処理速度を高速にするために，各ニブルの16進数の数字への変換には，命令シーケンス*ADD-DAA-ADC-DAA*を使用します(1ニブルは4ビットです)。

*HexStr*は，farコールで呼び出されるように書かれています。したがって*HexStr*は，Turbo Pascalのユニットの**インターフェース部**で宣言するか，または{ $ *F*＋}指令を指定してPascalのプログラムからfarコールで呼び出すようにしなければなりません。

```
CODE     SEGMENT
         ASSUME cs:CODE,ds:NOTHING

byteCount equ byte ptr ss:[bp+6]    ; パラメータ(push bpのために+2)
num       equ dword ptr ss:[bp+8]   ; パラメータ(push bpのために+2)
resultPtr equ dword ptr ss:[bp+12]  ; 関数の結果のアドレス(push bpのために+2)

HexStr    PROC FAR
          PUBLIC HexStr
          push bp                 ; 呼び出し側のベースポインタ
          mov  bp,sp              ; スタックポインタをBPにロード
          les di,resultPtr        ; 関数の結果を格納するアドレスをESとDI
                                  ; にロード
          mov dx,ds               ; PascalプログラムのDSをDXに退避
          lds si,num              ; 16進数に変換する数のアドレスをDS,SIにロード
          mov al,byteCount        ; バイト数をALにロード
          xor ah,ah               ; AHを0にする(ワードにするため)
          mov cx,ax               ; バイト数をCXにロード(CXでバイトを追跡)
          add si,ax               ; 数の最上位バイト(上位アドレス側)から開始
          dec si                  ; SIをデクリメント
          shl ax,1                ; AXを2倍する(16進数の桁数を求める)
          cld                     ; ディレクションフラグを0(増加)にセット
          stosb                   ; ALをES:DIの指すアドレスに格納
                                  ; (文字列の長さバイト)
HexLoop:
          std                     ; ディレクションフラグを1(減少)にセット
          lodsd                   ; DS:SIの指すアドレスからALにロード(次の1バイト)
```

```
        mov ah,al                       ; AHにコピーを保存
        shr al,1                        ; 上位ニブルだけにする
        shr al,1
        shr al,1
        shr al,1
        add al,90h                      ; ALに90hを加算
        daa                             ; パック形式のBCD(2進化10進数)に変換
        adc al,40h                      ; ALに40hとキャリビットを加算
        daa                             ; パック形式のBCDに変換(16進数の数字のASCII
                                        ; コード)
        cld                             ; ディレクションフラグを0(増加)にセット
        stosb                           ; ALをES:DIの指すアドレス(関数の結果の
                                        ; 格納領域)に格納
        mov al,ah                       ; 下位ニブルを16進数の数字に変換
        and al,0F
        add al,90h
        daa
        adc al,40h
        daa
        stosb
        loop HexLoop                    ; 数の全バイトが終了するまで繰り返す
        mov ds,dx                       ; PascalプログラムのDSを回復
        pop bp                          ; 呼び出し側のBPを回復
        ret 6                           ; パラメータの6バイトを破棄
HexStr  ENDP
CODE    ENDS
        END
```

上の*HexStr*を使用するPascalのプログラムは次のようになります。

```
program HexTest;
var
  num : word;
{$F+}
function HexStr (var num; byteCount : byte) : string; external;
{$L HEXSTR.OBJ}
{$F-}
begin
  num := $face;
  Writeln('The Converted Hex String is "',HexStr(num,sizeof(num)),'"');
end.
```

先のアセンブラルーチンをアセンブルし，上のPascalのプログラムに組み込んで実行するためのバッチファイルは次のようになります。

```
TASM HEXSTR
TPC HEXTEST
HEXTEST
```

.MODEL疑似命令を使うと，*HexStr*プログラムは次のようになります。

```
        .MODEL   TPASCAL
        .CODE

HexStr  PROC FAR num:DWORD,byteCount:BYTE RETURBNS resultPtr:DWORD
        PUBLIC HexStr
        les di,resultPtr          ; 関数の結果を格納するアドレスをESとDIにロード
        mov dx,ds                 ; PascalプログラムのDSをDXに退避
        lds si,num                ; 16進数に変換する数のアドレスをDSとSIにロード
        mov al,byteCount          ; バイト数をALにロード
        xor ah,ah                 ; AHを0にする(ワードにするため)
        mov cx,ax                 ; バイト数をCXにロード(CXでバイトを追跡)
        add si,ax                 ; 数の最上位バイト(上位アドレス側)から開始
        dec si                    ; SIをデクリメント
        shl ax,1                  ; AXを2倍する(16進数の桁数を求める)
        cld                       ; ディレクションフラグを0(増加)にセット
        stosb                     ; ALをES:DIの指すアドレスに格納(文字列の長さ
                                  ; バイト)
HexLoop:
        std                       ; ディレクションフラグを1(減少)にセット
        lodsd                     ; DS:SIの指すアドレスからALにロード(数の1バ
                                  ; イト)
        mov ah,al                 ; AHにコピーを保存
        shr al,1                  ; 上位ニブルだけにする
        shr al,1
        shr al,1
        shr al,1
        add al,90h                ; ALに90hを加算
        daa                       ; パック形式BCD(2進化10進数)に変換
        adc al,40h                ; ALに40hとキャリビットを加算
        daa                       ; パック形式BCDに変換(16進数字のASCIIコード)
        cld                       ; ディレクションフラグを0(増加)にセット
        stosb                     ; ALをES:DIの指すアドレス(関数の結果の格納領
                                  ; 域)に格納
        mov al,ah                 ; 下位ニブルを16進数の数字に変換
        and al,0F
        add al,90h
        daa
        adc al,40h
        daa
        stosb
        loop HexLoop              ; 数の全バイトが終了するまで繰り返す
        mov ds,dx                 ; PascalプログラムのDSを回復
        ret
HexStr  ENDP
CODE    ENDS
        END
```

先の*HexStr*のかわりに，この*HexStr*を使うことができます。Pascalのプログラムは，変更する必要はありません。バッチファイルコマンドも，変更する必要はありません。

2つの変数の交換

次の手続きは，サイズが*count*バイトの2つの変数を交換します。バイト数*count*が0のときは，64Kのデータを交換しようとします。

```
CODE      SEGMENT
          ASSUME cs:CODE,ds:NOTHING

var1      equ    DWORD PTR ss:[bp+12] ; パラメータ (push bpのためにオフセットが
var2      equ    DWORD PTR ss:[bp+8]  ; 2ずつ加算されていることに注意)
count     equ    WORD PTR ss:[bp+6]

Exchange  PROC FAR
          PUBLIC Exchange
          cld                       ; ディレクションフラグを0 (増加) にセット
          mov    dx,ds              ; DSをDXに退避
          push   bp               ; 呼び出し側のBPを退避
          mov    bp,sp              ; SPをBPにロード
          lds    si,var1            ; 第1の変数のアドレスをDSとSIにロード
          les    di,var2            ; 第2の変数のアドレスをESとDIにロード
          mov    cx,count           ; 転送するバイト数をCXにロード
          shr    cx,1               ; バイト数を2で割る(最下位ビットはCFに入る)
          jnc    ExchangeWords      ; バイト数が奇数でなければループに分岐
          mov    al,es:[di]         ; var2からALに1バイト読み込む
          movsb                     ; var1からvar2へ1バイト転送(SI,DIを+1)
          mov    [si-1],al          ; ALのバイトをvar1[SI-1]に書き込む
          jz     Finis              ; CXが0(変数が1バイト)なら交換終了
ExchangeWords:
          mov    bx,-2              ; BXに-2をロードしておく(ワード単位で
                                    ; 処理するため)
ExchangeLoop:
          mov    ax,es:[di]         ; var2から1ワードAXにロード
          movsw                     ; var1からvar2に1ワード転送 (SIとDIは+2)
          mov    [bx][si],ax        ; AXのワードをvar1[SI-2]に書き込む
          loop   ExchangeLoop       ; count/2回繰り返す
Finis:
          mov    ds,dx              ; PascalのDSを回復
          pop    bp
          ret    10
Exchange  ENDP
CODE      ENDS
          END
```

この*Exchange*手続きを使うPascalの例題プログラムは，次のようになります。

```
program TextExchange;

type
  EmployeeRecord = record
                     Name    : string[30];
                     Address : string[30];
                     City    : string[15];
                     State   : string[2];
                     Zip     : string[10];
                   end;
var
  OldEmployee, NewEmployee : EmployeeRecord;

{$F+}
procedure Exchange(var var1,var2; count : word); external;
{$L XCHANGE.OBJ}
{$F-}

begin
  with OldEmployee do
  begin
    Name := 'John Smith';
    Address := '123 F Street';
    City := 'Scotts Valley';
    State := 'CA';
    Zip := '90000-0000';
  end;
  with NewEmployee do
  begin
    Name := 'Mary Jones';
    Address := '9471 41st Avenue';
    City := 'New York';
    State := 'NY';
    Zip := '10000-1111';
  end;
  Writeln('Before: ',OldEmploye.Name,' ',NewEmployee.Name);
  Exchange(OldEmployee,NewEmployee,sizeof(OldEmployee));
  Writeln('After: ',OldEmployee.Name,' ',NewEmployee.Name);
  Exchange(OldEmployee,NewEmployee,sizeof(OldEmployee));
  Writeln('After: ',OldEmploye.Name,' ',NewEmployee.Name);
end.
```

先のアセンブラプログラムをアセンブルし，Pascalのプログラムに組み込んで実行するバッチファイルは次のようになります。

```
TASM XCHANGE
TPC XCHANGE
XCHANGE
```

.MODEL疑似命令を使うと，先のアセンブラプログラム*Exchange*は次のようになります。

```
        .MODEL   TPASCAL
        .CODE
Exchange  PROC FAR var1:DWORD,var2:DWORD,count:WORD
          PUBLIC Exchange
          cld                        ; ディレクションフラグを0(増加)にセット
          mov    dx,ds               ; DSをDXに退避
          lds    si,val1             ; 第1の変数のアドレスをDSとSIにロード
          les    di,var2             ; 第2の変数のアドレスをESとDIにロード
          mov    cx,count            ; 転送するバイト数をCXにロード
          shr    cx,1                ; バイト数を2で割る(最下位ビットはCFに入る)
          jnc    ExchangeWords       ; バイト数が奇数でなければループに分岐
          mov    al,es:[di]          ; var2からALに1バイト読み込む
          movsb                      ; var1からvar2へ1バイト転送(SI,DIを+1)
          mov    [si-1],al           ; ALのバイトをvar1[SI-1]に書き込む
          jz     Finis               ; CXが0(変数が1バイト)なら交換終了
ExchangeWords:
          mov    bx,-2               ; BXに-2をロードしておく(ワード単位で処理する
                                     ; ため)
ExchangeLoop:
          mov    ax,es:[di]          ; var2から1ワードAXにロード
          movsw                      ; var1からvar2に1ワード転送(SIとDIは+2)
          mov    [bx][si],ax         ; AXのワードをvar1[SI-2]に書き込む
          loop   ExchangeLoop        ; count/2回繰り返す
Finis:
          mov    ds,dx               ; PascalのDSを回復
          ret
Exchange  ENDP
CODE      ENDS
          END

          cld                        ; ディレクションフラグを0(増加)にセット
          mov    dx,ds               ; DSをDXに退避
          lds    si,val1             ; 第1の変数のアドレスをDSとSIにロード
          les    di,var2             ; 第2の変数のアドレスをESとDIにロード
          mov    cx,count            ; 転送するバイト数CXにロード
          shr    cx,1                ; バイト数を2で割る(最下位ビットはCFに入る)
          jnc    ExchangeWords       ; バイト数が奇数でなければループに分岐
          mov    al,es:[di]          ; var2からALに1バイト読み込む
          movsb                      ; var1からvar2へ1バイト転送(SI,DIはインクリメント)
          mov    [si-1],al           ; ALのバイトをvar1(SI-1)に書き込む
          jz     Finis               ; CXが0(変数が1 バイト)なら交換終了
ExchangeWords:
          mov    bx,-2               ; BXに-2をロードしておく(ワード単位で処理)
ExchangeLoop:
          mov    ax,es:[di]          ; var2から1 ワード AXにロード
          movsw                      ; var1からvar2に1ワード転送(SI,DIは2インクリメント)
          mov    [bx][si],ax         ; AXのワードをvar1(SI-2)に書き込む
          loop   ExchangeLoop        ; count/2回繰り返す
Finis:
          mov    ds,dx               ; PascalのDSを回復
          ret
Exchange  ENDP
CODE      ENDS
          END
```

先の*Exchange*のかわりにこの*Exchange*を使うことができます。Pascalのプログラムは変更する必要はありません。バッチファイルコマンドも変更する必要はありません。

DOSの環境変数の検索

次の*EnvString*関数は，DOSの環境変数の中から“*s*＝*SOMESTRING*”形式の文字列を検索し，見つかったら*SOMESTRING*を返します。

```
DATA      SEGMENT PUBLIC
          EXTRN prefixSeg : Word ; PSPのセグメントアドレス
DATA      ENDS
CODE      SEGMENT PUBLIC
          ASSUME cs:CODE,ds:DATA

EnvString PROC FAR
          PUBLIC  EnvString
          push    bp               ; 呼出し側のBPを退避
          cld                      ; 方向フラグを0(増加)にセット
          mov     es,[prefixSeg]   ; PSPのセグメント アドレスをESにロード
          mov     es,es:[2Ch]      ; PSPのオフセット 2Chの内容をESにロード
          xor     di,di            ; DIを0にする
          mov     bp,sp            ; SPをBPにロード
          lds     si,ss:[bp+6]     ; リターン アドレスの上のパラメータ(検索文字
                                   ; 列を指す
                                   ; ポインタ)をSIにロード
          ASSUME  ds:NOTHING
          lodsb                    ; [DS:SI]をALにロード(SIは+1)
          or      al,al            ; 長さバイトがゼロかどうかテスト
          jz      RetNul           ; ゼロならリターン
          mov     ah,al            ; ゼロでなければALをAHにコピーして保存
          mov     dx,si            ; SIをDXにコピーして保存
          xor     al,al            ; ALをゼロにする
Compare:
          mov     ch,al            ; 次のカウントのためにCHをゼロにセット
          mov     cl,ah            ; 長さバイトをCLにロード
          mov     si,dx            ; SIに検索文字列の先頭アドレスを再ロード
          repe    cmpsb            ; [DS:SI]と[ES:DI]をバイト単位で比較(比較後イン
          jne     Skip             ; クリメント)し，等しくなければ次の文字列を探す
          cmp    byte ptr es:[di],'='  ;等しければ次の文字が'='かテスト
          jne     NoEqual              ; '='でなければ探している文字列ではない
Found:
          mov     ax,es            ; ESを
          mov     ds,ax            ; DSにロード
          mov     si,di            ; DIをSIにロード(DS:SIに文字列のアドレスを
                                   ; ロード)
          inc     si               ; SIをインクリメントして'='をとおり越す
          les     bx,ss:[bp+10]    ; 関数の結果を格納するアドレスを
          mov     di,bx            ; ES:DIにロード
          inc     di               ; DIをインクリメントして長さバイトをとおり越す
          mov     cl,255           ; CLを文字列の最大長にセット
```

```
CopyLoop:
        lodsb                       ; [DS:SI]から1バイトAXにロードする(SIは+1)
        or      al,al               ; 長さバイトのゼロテスト
        jz      Done                ; ゼロなら終了
        stosb                       ; ALを[ES:DI]に格納する(DIは+1)
        loop    CopyLoop            ; コピーを繰り返す(最大255バイト)
Done:   not     cl                  ; CLをビット反転(コピーした文字列を求める)
        mov     es:[bx],cl          ; 関数結果の格納領域の先頭アドレスに長さを格納
        mov     ax,SEG DATA         ; PascalプログラムのグローバルDSをaxにロード
        mov     ds,ax               ; PascalプログラムのDSを回復
        ASSUME  ds:DATA
        pop     bp
        ret     4
        ASSUME  ds:NOTHING
Skip:
        dec     di                  ; DIをデクリメントしこのアドレス以後の文字を
                                    ; チェック

NoEqual:
        mov     cx,7FFFh            ; 最大検索範囲32KをCXにロード
        sub     cx,di               ; CXからDIを引き算して結果をCXに入れる
        jbe     RetNul              ; 領域の最後を越えていたら抜け出す
        repne   scasb               ; ALと[ES:DI]を比較し(ヌルが見つかるか,
                                    ; CXがゼロになるまで繰り返す).
        jcxz    RetNul              ; 見つからなければ抜け出す
        cmp     byte ptr es:[di],al ;ヌルが連続しているか
        jne     Compare                  ; ヌルでなければ次の文字列をテスト
RetNul:
        les     di,ss:[bp+10]       ; 関数の結果の格納アドレスをdiにロード
        stosb                       ; AL(ゼロ)を[ES:DI]に格納
        mov     ax,SEG DATA         ; PascalプログラムのグローバルDSをaxにロード
        mov     ds,ax               ; PascalプログラムのDSを回復
        ASSUME  ds:DATA
        pop     bp
        ret   4
EnvString ENDP
CODE    ENDS
        END
```

この*EnvString*関数を使うTurbo Pascalのプログラムは，次のようになります。

```
program EnvTest;
{ program looks for environment strings }
var
  EnvVariable : string;
  EnvValue : string;
{$F+}
function EnvString(s:string) : string; external;
{$L ENVSTR.OBJ}
{$F-}
begin
  EnvVariable := 'PROMPT';
  EnvValue := EnvString(EnvVariable);
  if EnvValue='' then EnvValue := '*** not found ***';
  Writeln('Environment Variable: ',EnvVariable,'  Value: ',EnvValue);
end.
```

先のアセンブラプログラムをアセンブルし，このTurbo Pascalのプログラムに組み込んで実行するためのバッチファイルは次のようになります。

```
TASM ENVSTR
TPC ENVTEST
ENVTEST
```

.MODEL疑似命令を使うと，アセンブラプログラム*EnvString*は次のようになります。

```
          .model   TPASCAL
          .DATA
          EXTRN prefixSeg : Word ; PSPのセグメントアドレス
          .CODE

EnvString PROC FAR  EnvVar:DWORD  RETURNS EnvVal:DWORD
          PUBLIC  EnvString
          cld                     ; ディレクションフラグを0(増加)にセット
          mov     es,[prefixSeg]  ; PSPのセグメントアドレスをESにロード
          mov     es,es:[2Ch]     ; PSPのオフセット2Chの内容をESにロード
          xor     di,di           ; DIを0にする
          mov     bp,sp           ; SPをBPにロード
          lds     si,EnvVar       ; リターンアドレスの上のパラメータ(検索文
                                  ; 字列を指すポインタ)をSIにロード
          ASSUME  ds:NOTHING
          lodsb                   ; [DS:SI]をALにロード(SIは+1)
          or      al,al           ; 長さバイトがゼロかどうかテスト
          jz      RetNul          ; ゼロならリターン
          mov     ah,al           ; ゼロでなければALをAHにコピーして保存
          mov     dx,si           ; SIをDXにコピーして保存
          xor     al,al           ; ALをゼロにする
Compare:
          mov     ch,al           ; 次のカウントのためにCHをゼロにセット
          mov     cl,ah           ; 長さバイトをCLにロード
          mov     si,dx           ; SIに検索文字列の先頭アドレスを再ロード
          repe    cmpsb           ; [DS:SI]と[ES:DI]をバイト単位で比較(比較後
                                  ; インクリメント)し，等しくなければ次の文字列
          jne     Skip            ; を探す
          cmp    byte ptr es:[di],'=' ; 等しければ次の文字が'='かテスト
          jne     NoEqual             ; '='でなければ探している文字列ではない
Found:
          mov     ax,es           ; DS:SIが見つかった文字列のアドレスを指すように
          mov     ds,ax           ; ESをDSにロード
          mov     si,di           ; DIをSIにロード
          inc     si              ; SIをインクレメントして'='をとおり越す
          les     bx,EnvVal       ; 関数の結果を格納するアドレスを
          mov     di,bx           ; ES:DIにロード
          inc     di              ; DIをインクリメントして長さバイトをとおり越す
          mov     cl,255          ; CLを文字列の最大長にセット
CopyLoop:
          lodsb                   ; [DS:SI]から1バイトAXにロードする(SIは+1)
          or      al,al           ; 長さバイトのゼロテスト
          jz      Done            ; ゼロなら終了
          stosb                   ; ALを[ES:DI]に格納する(DIは+1)
          loop    CopyLoop        ; コピーを繰り返す(最大255バイト)
```

```
Done:       not     cl              ; CLをビット反転(コピーした文字列を求める)
            mov     es:[bx],cl      ; 関数結果の格納領域の先頭アドレスに長さを格納
            mov     ax,SEG DATA     ; Pascalプログラムのグローバル DSをaxにロード
            mov     ds,ax           ; PascalプログラムのDSを回復
            ASSUME  ds:DATA
            ret
            ASSUME  ds:NOTHING
Skip:
            dec     di          ; DIをデクリメントし,このアドレス以後の文字をチェック
NoEqual:
            mov     cx,7FFFh        ; 最大検索範囲32KをCXにロード
            sub     cx,di           ; CXからDIを引き算して結果をCXに入れる
            jbe     RetNul          ; 領域の最後を越えていたら抜け出す
            repne   scasb           ; ALと [ES:DI]を比較し(ヌルが見つかるか,


                                    ; CXがゼロになるまで繰り返す).
            jcxz    RetNul          ; 見つからなければ抜け出す
            cmp     byte ptr es:[di],al  ; ヌルが連続しているか
            jne     Compare              ; ヌルでなければ次の文字列をテスト
RetNul:
            les     di,EnvVal       ; 関数の結果の格納アドレスをdiにロード
            stosb                   ; AL(ゼロ)を [ES:DI]に格納
            mov     ax,SEG DATA     ; Pascalプログラムのグローバル DSをaxにロード
            mov     ds,ax           ; PascalプログラムのDSを回復
            ASSUME  ds:DATA
            ret
EnvString ENDP
CODE        ENDS
            END
```

先の*EnvString*のかわりに，この*EnvString*を使うことができます。Pascalのプログラムは，変更する必要はありません。バッチファイルコマンドも，変更する必要はありません。

第8章

Turbo AssemblerとTurbo Basicのインターフェース

Turbo Assemblerは，Microsoft社のMacro Assemblerに対して上位互換性がありますので，Turbo Basicでプログラムを組んでいる人も，Turbo Assemblerのルーチンを使用することに特に問題はありません。この章ではTurbo Basicのマニュアルに記載されているTurbo Basicの例題プログラムのいくつかと新しい例題プログラムを用いて，Turbo AssemblerによりTurbo Basicの能力を拡張する方法を説明します。

注意：Turbo Basicは，特に断わらないかぎり，バージョン1.0およびそれ以降のバージョンです。

Turbo Basicでは，次の方法によりBasicのプログラムの内部でアセンブラルーチンを実行することができます。

- ■CALLによりアセンブラルーチンを呼び出す。
- ■CALL ABSOLUTEによりアセンブラルーチンを呼び出す。
- ■CALL INTERRUPTとTurbo Basicの割り込み処理サポート機能によりアセンブラルーチンに分岐する。

どの呼び出し方法を使う場合にも，いくつかのレジスタの値を退避しなければなりません。CALL INTERRUPTは保存しなければならないレジスタが最も少なく，SS（スタックセグメント）とSP（スタックポインタ）を退避するだけで済みます。他の2つの方法では，SSとSPだけでなく，DS（データセグメント）とBP（ベースポインタ）を退避しなければなりません。

レジスタを退避しなければならないということは，必らずしも退避しなければ破壊されることを意味しません。簡単なルーチンでは，どのレジスタも変更されないかもしれません。したがってレジスタも退避する必要はないかもしれません。"変更されない"ではなく"変更ないかもしれない"というのは，変更されないと考えることは避けたほうがいいから

です。アセンブラのプログラミングでは特にそうです。MS-DOSのマニュアルに，ある割り込みがスタックポインタやベースポインタ（またはその他のレジスタ）の内容を変えないと書いてあるからといって，いつでもそうとは限りません。MS-DOS自体が変わることがあり，また割り込みの組み合わせによってはマニュアルの記載と異なることがあります。このような状況のもとでは，安全策を取るほうが賢明です。必要なレジスタを退避するために要する損失は，新しくリリースされるMS-DOSに対する移植性と退避しない場合のリスクを比較すると取るに足りません。

パラメータの受け渡し

Turbo Basicは，パラメータをスタックに積んでアセンブラのルーチンに渡します。この呼び出しはすべてfarコールです。したがってスタックの最後の4バイトは，アセンブラルーチンからリターンするときのリターンアドレスです。リターンアドレスのすぐ上（上位アドレス）のパラメータ（第1のパラメータ）のアドレスは［SP+4］になります。アセンブラルーチンでレジスタをスタックにプッシュするときは，各パラメータのアドレスに，1レジスタあたり2ずつ加算しなければなりません。スタックは，下位メモリに向って伸びることを忘れないでください。

すべての単純変数（配列以外の変数）は，セグメント（2バイト）とオフセット（2バイト）で渡され，スタック上で4バイトを占めます。これは後で説明するように重要な利点です。

定数や式のように値で渡されるパラメータも，スタック上で4バイトを占めます。スタックに積まれる値は式の値ではなく，式の値が一時的に格納されているアドレスです。このやり方は一見遠回りのように見えるかもしれませんが，2つの利点があります。第1に，すべてのアセンブラルーチンは，渡される値をまったく同じ方法で処理することができることです。第2に，変数で渡すべきパラメータを間違えて値で渡しても，パラメータを変更するルーチンがメモリの重要な領域の内容を変更する心配がないことです。

このパラメータの受け渡し方法により，アセンブラルーチンを標準化することが一層容易になります。またこの受渡し方法には，さらに次のような利点があります。たとえば整数変数*x%*の値が4であり，整数を渡すアセンブラルーチン*MYROUTINE*があるとしましょう。このルーチンは，CALL MYROUTIE (x%) で呼び出しても，CALL MYROUTINE (4%) で呼び出しても，まったく同じように動作します。このルーチンが渡されたパラメータの値を変更するとき，CALL MYROUTINE（4%）で呼び出すと，整数値4が一時的に格納されているアドレスの内容を変更しますが，害はありません。

CALL MYROUTINE (4%) において，データの型が明示されていることに注意してください。型を明示することは必らずしも必要ではありませんが，良い習慣です。型を明示しないと，Turbo Basicがこの値4を単精度数と見なした場合には，アセンブラルーチンは間違った値（4バイトの単精度数の2バイト）を取得することになります。アセンブラルーチンに確実に正しい型のパラメータを渡すには，ルーチンを呼び出すときにいつでも変数や値の型を明示するようにするべきです。

アセンブラルーチンに配列を渡すときは，スタックは60バイト伸びます。渡される情報のほとんどは，そのルーチンに必要のない情報です。Turbo Basicのマニュアルには，配

列全体を渡すかわりに，必要な配列のパラメータだけを整数として渡すように奨められています。配列の全体ではなく少数の選択したパラメータだけを渡すことで，スタックスペースを節約することができます。またアセンブラのルーチンを呼び出すのに必要な時間を短縮することができます。さらにTurbo Basicの将来のバージョンに対する移植性が向上します。

たとえば，ベースポインタBPだけをスタックに退避すればよい簡単なルーチンがあるとしましょう。この場合には，リターンアドレスのすぐ上（上位アドレス）の第1のパラメータのスタックアドレスは［*SP*+*6*］になります。アセンブラルーチンで2つのレジスタをプッシュする場合は［*SP*+*8*］になります。

第1のパラメータが整数の定数で，アセンブラルーチンに渡される（定数方式）としましょう。この場合には，次のように書くだけでこの整数の値をCXレジスタに入れることができます。

```
push   bp              ; BPを退避
mov    bp,sp           ; BPがスタックトップを指すようにする
les    di,[bp+6]       ; 値のセグメントアドレスをESに，オフセットをDIにロード
mov    cx,ES:[DI]      ; ES:DIが指すアドレスの値をCXにロード
```

注意：定数は，通常の変数とは別のセグメントにあります。したがって定数にアクセスするには，正しいフルアドレスを使うように気をつけてください。カレントデータセグメント以外の場所にある変数については，後でさらに説明します。

これに対して整数の変数が渡される（変数方式）ときは，［*BP*+*6*］はデータセグメント内の変数のオフセットアドレスです。この整数変数の値をCXレジスタにロードするには，次のように書くことができます。

```
push   bp              ; BPを退避
mov    bp,sp           ; BPがスタックトップを指すようにする
mov    bx,[bp+6]       ; 変数のオフセットをBXにロード
mov    cx,[bx]         ; BXの指すアドレスの値(変数の値)をCXにロード
```

このルーチンでは，変数はカレントデータセグメント内にあり，データセグメント内のオフセットだけわかれば変数の値にアクセスできると仮定しています。

アセンブラルーチンを書くときは，変数は変数方式でなく定数方式で渡されると考えるほうが安全です。そうすれば変数が実際には変数方式で渡されても，何も問題はありません。スタックに積まれるフルアドレスのセグメントアドレスがカレントセグメントになるだけです。反対に，変数方式で引用しなければならない変数が定数方式で引用されると，オフセットしか取得できないためセグメントアドレスが正しくありません。そのためこのルーチンは，間違ったメモリ領域の値を読み出したり変更したりして，予測できない結果

を生じることがあります。

文字列，配列，浮動小数点数のような変数に対しては，変数を変数方式で渡すほうがずっと簡単です。これらの変数は長いので，もし実際にスタックに積んで渡すといろいろな問題が起きます。さらにスタックから長い変数を読み出すには，スタックから変数のアドレスを取得してそのアドレスにある変数を操作するのと同じくらい，オーバーヘッドが大きくなります。また文字列変数に対しては，非常に短かい文字列を除いて，メモリアクセスを行なわずに文字列を処理することができるだけのレジスタスペースはありません。

カレントデータセグメント外の変数

渡される変数がカレントデータセグメント内にないときは，アセンブラルーチンの内部でその変数の値にアクセスするには，変数のオフセットとセグメントの両方が必要です。Turbo Basicはいつでも変数のセグメントとオフセットの両方をスタックに積みます。したがってアセンブラルーチンにおいていつでも，変数のフルアドレスを取得することができます。

アドレスのセグメント部は，パラメータのオフセットのすぐ上の2バイトに入っています。アセンブラプログラムでこの情報を利用するための最も便利な方法は，**LES**命令を使うことです。

LES命令は，ESレジスタにアドレスのセグメント部をロードし，オペランドに指定されたレジスタに変数のオフセット値をロードします。この命令を使うと，変数がどのセグメントにある場合にも，変数のフルアドレスを使うことが保証されます。

もう一度，整数変数の値をCXレジスタにロードするアセンブラルーチンを取り上げましょう。

```
push  bp              ; トップを指すようにする
mov   bp,sp           ; BPがスタック
les   di,[bp+6]       ; 変数のセグメントをESに，オフセットをDIにロード
mov   cx,ES:[DI]      ; ES:DIの指すアドレスの値(変数の値)をCXにロード
```

Turbo Basicはいつでも変数のフルアドレスを渡すので，データが格納されているセグメントによらないアセンブラルーチンを書くことができます。Turbo Basicのプログラムを書き換えて変数や配列を別のセグメントに入れても，アセンブラルーチンがフルアドレスを使って変数にアクセスするようになっていれば，アセンブラルーチンを書き換える必要はありません。

どのCALLを使うべきか

*far*と*near*の2種類のCALLがあります。far CALLはカレントコードセグメント外のルーチンを呼び出すことができます。near CALLは，カレントコードセグメント内のルーチンしか呼び出せません。

CALL ABSOLUTEは，問題を引き起こす可能性があります。なぜならCALL ABSOLUTEは，メモリのどのアドレスでも呼び出すことができるからです。Turbo Basicは，CALL ABSOLUTEルーチンに制御を渡すときは，自動的にfarコールを生成します。したがってこのルーチンは，いつでもfarリターンで終了しなければなりません。

CALL INTERRUPTは暗黙にfarコールを生成することができますが，Turbo Basicはそれを内部的に処理します。ユーザが自分で割り込みハンドラを書くときは，**IRET**(割り込みからのリターン）命令だけでBasicのプログラムに制御に戻すことができます。

インラインアセンブラは，Turbo Basicのプログラムをコンパイルするときに，Basicのプログラムに挿入されます。アセンブラルーチンのコードは，通常コードセグメント内に入れられますが，いつでもそうとは限りません。このルーチンもfarリターンで終了します。ただしTurbo Basicは，自動的に呼び出しとリターンを行ないますので，アセンブラルーチンの中に**RET**命令を使わないでください。コードの途中でルーチンを終了したい場合は，コードの最後のラベルにジャンプさせてください。

注意：Turbo BasicはDOS LINKプログラムを使わないので，ルーチンを**PUBLIC**と宣言したり，Basicのプログラムの内部でそれらのルーチンをexternalと宣言する必要はありません。

スタックのポップ

ルーチンが終了する前に，スタックにプッシュしたすべてのレジスタを，スタックからポップしなければなりません。ルーチンが条件によりレジスタを**プッシュ**，**ポップ**したりしなかったする場合は，特に間違えやすいので注意してください。

Turbo Basicは，スタックのトップがリターンすべきアドレスであると見なします。したがってポップするレジスタが少なすぎても多すぎても，呼び出し終了後Basicのプログラムに復帰できません。

セグメントレジスタに無意味な値をロードしたりポップしたりしないでください。そのような操作を含むソースコードは，DOSの将来のバージョン（たとえばプロテクトモードのOS/2）で動作しないかもしれません。

アセンブラプログラムの作成

アセンブラのプログラムを書いて，Turbo Basicのプログラムの中で使用するためにそれを.COMファイルに変換したいときは，Turbo Basicのマニュアルに記載されている次のバッチファイルを使うことができます。

```
TASM %1;
TLINK /t %1;
```

アセンブラルーチン側ではスタックセグメントは設定しないでください。何故ならこのアセンブラルーチンは，Turbo Basicにより用意されたスタックを使うからです。

Turbo Assemblerは，プログラムの先頭に**ORG 100h**文を置かなくても，デフォルトで開始アドレスを100hにします。しかし後で見るときにわかりやすいように，**ORG**により明示的に指定しておくほうが良いやり方です。

80186, 80286,または80386プロセッサで実行するアセンブラルーチンは，コードの先頭で**.186, .286,**または**.386**疑似命令を使ってプロセッサを指定することができます。プロセッサを指定すると，そのプロセッサのすべての命令が使用できます。後で説明するように，これは非常に大きな利点です。

インラインアセンブラプロシージャの呼び出し

アセンブラルーチンを書き，Turbo Assemblerでそれを.COMファイルに変換したとしましょう。このルーチンをTurbo Basicの内部で使用する方法は2つあります。＄INLINE COM疑似命令を使う方法と＄INCLUDE指令を使う方法です。

最も直接的な方法は"**$INLINE COMファイル名**"を使う方法です。Turbo Basicは，指定された場所に指定された.COMファイルを挿入します。この方法は比較的簡単ですが，いくつか不利な点があります。

- ■Turbo Basicでは，1プロシージャあたり16個までの＄INLINE指令しか指定できません。何かかなり複雑なことをするときは，この制約が問題になる可能性があります(ただしごくまれでしょうが)。
- ■.COMファイルにはコメントや説明がありません。もちろん呼び出し側のプログラムに，REM文により.COMファイルの説明を入れることはできます。しかし.COMファイル自体にコメントや説明を入れることができればもっと便利です。
- ■＄INLINE .COMファイルは，ファイルの数が多くなる可能性があります。いくつかの＄INLINE .COMファイルを，1つのファイルにまとめて入れておくことができれば便利です。複数の＄INLINE .COMファイルをよく一緒に使うときは特に便利でしょう（アセンブラルーチンのライブラリを使う理由の1つはこれですが，.COMファイルのライブラリを作成するのは容易ではありません)。
- ■INLINE .COMファイルを変更するには，ソースを変更してアセンブルし直さなければなりません。変更が簡単なときは，これは面倒です。

＄INLINE .COMには上のような問題があるので，.COMファイルを，＄INCLUDE疑似命令を使ってプログラムに挿入できる16進数のシーケンスに変換したいことがあるかもしれません。16進数のシーケンスに変換したルーチンは，Turbo BasicのエディタのRead Fileコマンド（***CTRL-K R***）でディスクから読み込むこともできます。このようにすると，ソースファイルにインクルードされるルーチンを見ることができます。Turbo Basicでプログラムを書くとき，これは非常に便利です。

16進数のコードは編集可能なテキストですから，コメントを付け加えることができます。またインラインコードの簡単な変更は，Turbo Basicのエディタを使って行なうことができます。再アセンブルする必要はありません。さらに複数のルーチンを1つのファイルに入れることもできます。これらのテクニックを組み合わせると，アセンブラルーチンのライ

ブラリを作成することができます。このようなライブラリのほうが，本格的なライブラリマネジャを使うよりも管理が容易でしょう。

ルーチンが非常に長いときは，16進数コードのファイルが非常に大きくなり，ソースファイルの編集が困難になることがあります。編集できるファイルの大きさは，最大64Kbです。ファイルの大きさが問題になるときは，16進数ファイルを $INCLUDEファイルとしてBasicのプログラムに取り込みます(そのように大きなルーチンがプログラムをわかりやすくすることはありませんから，ソースプログラムに読み込む意味もありません)。

次に.COMファイルを16進数コードのファイルに変換するTurbo Basicのプログラムを示します。

```
'COM2INC.BAS
'このプログラムはCOMファイルを$INCLUDEファイルに変換する．

DEFINT  A-Z
'変数はすべて整数

F$=COMMAND$
'コマンドラインを調べる

WHILE F$=""
  PRINT "このプログラムはCOMファイルを$INCLUDEファイルに変換します．ソース"
  PRINT "ファイルのデフォルトのファイルタイプはCOMです．出力ファイルのデフ"
  PRINT "ォルトのファイルタイプはINCです．ファイルタイプを指定することによ"
  PRINT "りデフォルトのファイルタイプをオーバーライドすることができます．出力"
  PRINT "ファイルの名前を指定しないときは，入力ファイルと同じ名前で，ファイル"
  PRINT "タイプがINCになります．"
  LINE INPUT "変換するファイルの名前を入力してください： ";F$
WEND

IF COMMAND$="" THEN
  LINE INPUT "出力ファイルの名前を入力してください： ";O$
END IF

IF INSTR(F$,".")=0 THEN F$=F$+".COM"      '入力ファイル名に.COMを付加
IF O$="" THEN
  O$=LEFT$(F$,INSTR(F$,"."))+"INC"        '入力ファイル名に.INCを付加して
                                          '出力ファイル名にする
  ELSE
    IF INSTR(O$,".")=0 THEN O$=O$+".INC" '入力された出力ファイルに.INCを付加
END IF

OPEN "R",#1,F$,1                        '入力ファイルをオープン
FIELD #1, 1 AS A$                       '入力ファイルからA$に1バイトずつ読み込む

LASTBYTE&=LOF(1)                        'ファイルエンド
OPEN "O",2, O$                          '出力ファイルをオープン
```

```
FOR I&=1 TO LASTBYTE&-1
  GET 1,I&
  X%=ASC(A$)
  IF ((I&-1) MOD 5=0) THEN PRINT #2,"": PRINT #2,"$INLINE";
  PRINT #2,"&H"; HEX$(X%);
  IF ((I&-1) MOD 5<>4) THEN PRINT #2,",";
NEXT I&
GET 1,LASTBYTE&
PRINT #2,"&H"; HEX$(ASC(A$))
PRINT "変換が完了しました。"; LASTBYTE&; "バイト読み込みました."
PRINT O$; "は"; LOF(2); "バイトです."
CLOSE
END
```

このプログラムの出力ファイルは，各行の16進数コードの数が5以下に設定されています。各行は＄INLINE疑似命令で始まり，行の右側にはコメントを追加する十分なスペースがあるはずです。MOD 5をMOD N（Nは5以外の数）に変更するだけで，各行の最大コード数を増減することができます。

16進数コードに変換したルーチンを変更したいときは，変更が僅かで十分詳細な説明があれば，ソースファイルを変更して再アセンブルしなくても，直接16進数コードを変更することができます。

メモリの中のTurbo Basicのルーチンを探す

ルーチンのアドレスを見つけるために，次の3つの方法が考えられます。

■ルーチンが自分でアドレスを返すようにする。

■複数のルーチンを一緒にまとめて，そのうちの1つのルーチンがアドレスを返すようにする。

■ユーザが自分で目印となる特別のバイト列を探す。

ルーチンがアドレスを返すようにするには，次のようなコードを使うことができます。

```
xy:  mov  ax,cs            ; コードセグメントレジスタをAXにコピー
     push bp               ; BPを退避
     mov  bp,sp            ; BPがスタックトップを指すようにする
     les  di,[bp+6]        ; 第1のパラメータをES,DIにロード
     mov  es:[di],ax       ; カレントコードセグメントを[ES:DI]に格納
     mov  dx, offset xy    ; xyのオフセットをdxにロード
     les  di,[bp+0ah]      ; 第2のパラメータをES,DIにロード
     mov  es:[di],dx       ; xyのオフセットを[ES:DI]にロード
     jmp  fin              ; ジャンプしてコード本体を迂回
                           ; ここにコード本体が入る
fin: pop bp                ; BPを回復してリターン
```

このルーチンには，2つのパラメータを渡す必要があります。このルーチンは，第1のパラメータが指すアドレスにコードセグメントのアドレスを，第2のパラメータが指すアドレスにオフセットを返します。問題は，このコードは一度使用されたら不要になることです。実際には，無駄であるよりさらに悪いと言うべきでしょう。このコードを取り除かないとルーチン本体が正常に実行されないからです。

ただしこのコードを取り除くことによりルーチンの速度が大幅に向上しなければ，変更する意味がありません。またルーチンが完全に再配置可能でないと，実際のコードの前に多数のNOPが入ります。

上の方法では，とにかくルーチンのアドレスを求めることはできます。それに対して，複数のルーチンを一緒にまとめてTurbo Basicのプログラムからラベルによりその中のルーチンを呼び出すようにしたときは，それらのルーチンの中にアドレスを返すルーチンを含めることができるでしょうか。

答えはノーです。Turbo Basicは，ユーザにかわってリターン処理をすることを思い出してください。各ルーチンには異なる名前がつけられるので，Turbo Basicは，各ルーチンが再配置可能なコードであると見なします。そのため最終の.EXEファイルにおいて，す

べてのルーチンが同じメモリ領域に入ることは保証できません。またたとえメモリの同じ領域に同じ順番で入れられていても，Turbo Basicがそれらのルーチンの間に何バイトのコードを挿入したかわかりません。そのため変更したいルーチンがどこにあるかわかりません。

ルーチンのアドレスを決定する3番目の方法は，変更したいルーチンの**目印**となるバイト列を探す方法です。

この方法にも問題があります。第1に，このような検索はかなり時間がかかることです。第2に，一致するバイト列が見つかっても，それが探しているルーチンであるという保証はありません。第3に，各ルーチンに目印となる異なるバイト列を入れなければならないことです。そのためメモリが無駄になり，速度も低下します。

プログラムが変更できるルーチンを作成するには，ルーチンのアドレスを決定するもっと良い方法が必要です。またルーチンの中の命令を，簡単に変更できる方法が必要です。

これらの問題を解決する方法については，次の節を読んでください。次の節では，Turbo Basicのプログラムの内部から変更できるルーチンを使うための特殊な方法を説明します。

文字列を隠す

Turbo Basicでは，文字列変数に最大64Kbの領域を確保することができます。ときにはこの領域の全バイトが必要になることがあるうえに，メニューやプロンプトの表示に使用する引用符で囲んだ文字列定数も，この文字列領域を使います。

しかしコード領域は，最大16セグメント（各セグメントは最大64Kb）もあります。この広いコード領域に文字列定数の一部を格納することができれば，動的文字列データのための領域を減らさなくても済みます。好都合なことに，文字列定数をコード領域に格納することは，それほど難しくはありません。次のルーチンを見てください。

```
; このルーチンは2つの整数パラメータを受け取り，プログラム本体の中のテキストの
; セグメントとオフセットを返します.
;
      push   bp
      mov    bp,sp
      mov    dx,offset show    ; 文字列のオフセットをDXにロード
      mov    ax,cs             ; コードセグメントをAXにロード
      les    di,[bp+6]         ; 第1のパラメータをES:DIにロード
      mov    es:[di],dx        ; 文字列のオフセットを[ES:DI]に格納
      les    di,[bp+0ah]       ; 第2のパラメータをES:DIにロード
      mov    es:[di],ax        ; コードセグメントを[ES:DI]にロード
      jmp    fini              ;
      DB     'ここに文字列を入れる。どんな文字列でもよい.'
      DB     '文字列の最後に入れる文字は自由に決めることが'
      DB     'できる. ここではヌル.',0
fini  pop    bp
```

このルーチンの効果は，先に提案したプログラムにより変更可能なインラインコードとは多少異なります。Turbo Basicは，このルーチンをアセンブルした.COMファイルを全体がコードであるかのように扱いコード領域に格納しますが，それによりデータがコード領域に格納されます。

このルーチンは，内部に挿入されているデータのカレントアドレスを返します。データの長さも知りたいときは，長さも返すようにすることができます。その場合は，このルーチンに，3つめの整数パラメータを渡さなければなりません。

これでテキストがメモリのどこにあるかわかりるので，文字列を表示したいときはいつでも，PEEKを使ってその文字列データを文字列領域に読み込むことができます。メッセージの表示が終了したら，不要になった文字列を捨てることができます。再び必要になったら，コード領域から読み出してくることができます。

このルーチンの中の上書き可能なバイト数，特にテキストより前の部分のバイト数は，計算することができます。そして最後の命令（jmp fini）以外のすべての命令を，ユーザのコードで置き換えます。ルーチンの場所と大きさがわかっていますので，BLOADを使って上書きすることができます。Turbo Basicの.EXEファイルに関するかぎり，ルーチン全体を変更しても何も変化しません。

通常この手法は必要ありませんが，コード領域に文字列を格納する方法は役に立つことがあります。ただしルーチン全体を別のルーチンで置き換えるには，CALL ABSOLUTEを使ったほうが便利です。

CALL ABSOLUTE

Turbo Basicのマニュアルには，CALL ABSOLUTEについて簡単な説明しかありません。その理由は，第1に，Turbo Basicはこのルーチンをあまりよく制御できないからです。第2に，このルーチンはBasicインタプリタ用に書かれていたので，普通に使用されているからです。ただしTurbo BasicはBasicインタプリタとかなり異なっているので，それらのルーチンは動作しないかもしれません。第3に，将来のオペレーティングシステムでは，CALL ABSOLUTEルーチンが使用できないかもしれないからです。特にコード領域とデータ領域を厳格に分離するオペレーティングシステムでは，データ領域にある命令を実行することはできないかもしれません。第4に，CALL ABSOLUTEで呼び出すルーチンには，単純な整数変数しか渡せないからです。これは，実際にはそれほど大きな制約ではありません。なぜなら2つの整数変数によりセグメントアドレスとオフセットアドレスを渡すことにより，すべての型の変数を渡すことができるからです。しかしこの方法は，パラメータの受け渡しに多少時間がかかります。

この説明では，MS-DOSのバージョン2.0かそれ以降のバージョンを使用し，オペレーティングシステムはメモリのどこにある命令でも実行を許容するものとしています。

固定アドレスに対するCALL ABSOLUTE

複数のプログラムが同じ1組のルーチンを使用するとき，それらのルーチンを決まった領域に置くことは理にかなっています。そうすれば，それらのルーチンを使う必要のあるプログラムは，決まったアドレスにより各ルーチンを呼び出すことができます。Turbo Basicでは,MEMSETを使って，この目的のために高位メモリに領域を確保することができます。

ENDMEMは，よくMEMSETと一緒に使用されます。ENDMEMは，コンパイルされたTurbo Basicのプログラムが使用できるメモリの最後のアドレスをlong型の整数で返します。これらのルーチンは通常,この上限アドレスの下から始まる上位メモリの固定領域に置かれます。

このようなルーチンは,CALL ABSOLUTEで呼び出す必要があるでしょう。これらのルーチンをメモリに読み込むにはBLOADを使います。BLOADには,ロードするオフセットアドレスを指定しなければなりません。セグメントのアドレスはDEF SEGで指定します。

Turbo Assemblerでこのようなルーチンを作成するときには，次のルールを守らなければなりません。

1. ルーチンがどのアドレスでも動作するようにしたいときは,そのルーチンの中の分岐（**JMP**命令）や呼び出し（**CALL**命令）が完全に最配置可能でなければなりません。
2. ルーチンが特定のアドレスだけ動作するようにしたいときは,アセンブラソースコードの先頭で,**ORG**疑似命令によりそのアドレスを指定しなければなりません。

変動アドレスに対するCALL ABSOLIUTE

Turbo Basicでは，プログラムを実行するたびに変化する可能性のあるメモリアドレスに対してもCALL ABSOLUTEを使うことができます。その代表例は，通常データ領域外の配列にアセンブラルーチンを読み込む場合です。次のコードを見てください。

```
DEFINT A-Z
$DYNAMIC                                    '動的配列
DIM ROUTINEARRAY(10000)                     '20,002バイトを割り当てる
'ここにいろいろなコードが入る
WHERESEG%=VARSEG(ROUTINEARRAY(0))           '配列のセグメントアドレスを取得
WHEREOFFSET%=VARPTR(ROUTINEARRAY(0))        '配列のオフセットアドレスを取得
DEF SEG=WHERESEG%                           '配列のセグメントをセット
BLOAD"COMFILE",WHEREOFFSET%                 'ルーチンを読み込む
CALL ABSOLUTE WHEREOFFSET%(PARAMETER%)      'ルーチンを呼び出す
DEF SEG                                     'デフォルトセグメントに戻す
```

複数のルーチンを使用したいときは，同じ配列の中に各ルーチンを順番にロードすることもできます。またあるルーチンの複数のバージョンを使いたいときは，ロードするバージョンを選択して，それらのバージョンを異なる配列にロードすることもできます。ルーチンを部分的に変更したいときは，配列の要素の値を変更するだけで済みます。

CALL ABSOLUTEにより使用するようにしたルーチンは，$INLINEで挿入するルーチンよりも，アドレスを見つけ出して変更することが容易です。CALL ABSOLUTEルーチンの問題点は，ルーチンが広く使用できるためには，完全に最配置可能でなければならないことです。短かいルーチンではこのことは問題にならないかもしれませんが，複雑なルーチンでは完全に再配置可能なコードを書くことは非常に難しくなります。

またBLOADにより，ルーチンを文字列変数に読み込むこともできます。この場合には特に注意深くしなければなりません。BLOADにより文字列変数よりも長いルーチンを読み込んだら，他の文字列の上に上書きするおそれがあります。この場合には上書きされた文字列が変更されると，BLOADで読み込んだルーチンの一部が変更されます。

文字列変数もアドレスが動く可能性があります。したがってルーチンが文字列に正しくロードされたときも，そのルーチンを呼び出す直前に，VARSEGとVARPTRを使って文字列のアドレスを求める必要があります。

Turbo Basicの文字列は，数値変数とは異なる方法で格納されます。VARPTR（A%）を実行すると，整数変数*A%*のアドレスが得られます。またVARPTR（A%）を実行すると，文字列記述子*A$*のアドレスが得られます。*A$*のアドレスよりさらに2バイト上のアドレスには，文字列領域内の文字列の実際のアドレスが入っています。したがって文字列変数のアドレスを求めるには，次のようにしなければなりません。

```
A%=VARPTR(A$)
A%=A%+2
STRINGADDRESS%=CVI(CHR$(PEEK(A%))+CHR$(PEEK(A%+1)))
```

アセンブラルーチンを文字列に読み込むことは広く行なわれていましたが，現在では整数配列は宣言したり削除したりできるようになっているので，配列に比べてうまい方法ではなくなりました。CALL ABSOLUTEルーチンは整数配列に読み込むようにし，アドレスが絶えず変化する可能性のある文字列データにアクセスするための余計な困難は避けたほうがいいでしょう。

BLOADを使うことを避けたいときは，バイナリファイルI/Oを使って.COMファイルを文字列にロードすることもできます。すなわち .COMファイルをバイナリタイプとしてオープンし，文字列の中に正しいバイト数を読み込みます。またこの方法を使って，整数配列にデータを読み込むこともできます。ただしBLOADのほうが速くて簡単です。

CALL ABSOLUTEのその他の問題点

ディスクから読み込んでCALL ABSOLUTEで呼び出して使うためのコードには，いくつか問題があります。最も大きな問題は，先に説明したように，再配置可能でなければならないことです。

もう1つの大きな問題は，このようなルーチンはメインプログラムとは別にディスクから読み込まなければならないこと，そのためにエラーが発生する可能性が増すことです。たとえば必要なコードがディスクになかったり，あっても壊れていたりすることがあるからです。

3つ目の問題点は，ディスクからルーチンを読み出すためにあまり時間がかかると，アセンブラルーチンを使う意味がなくなることです。

このような問題点はありますが，CALL ABSOLUTE構造には，異なるルーチンを読み込むことができること，プログラムの制御のもとに変更できるルーチンを使うことができること，メモリに常駐させる必要のあるコードの量を減少させることができることなどの大きな利点があります。これらの利点は，CALL ABSOLUTEルーチンの使用を考えてみる十分に強力な理由です。

CALL INTERRUPT

Turbo Basicのプログラムの内部からアセンブラルーチンにアクセスする3番目の，そして最後の方法は，アセンブラを避ける最も簡単な方法であり，またアセンブラを使用する最も難しい方法でもあります。

ほとんどのプログラマは，CALL INTERRUPTを使って，通常のMS-DOSのサービスにアクセスします。この方法ならアセンブラは使わなくて済みます。そのかわり，次のことを憶えておかなければなりません。

名　前	レジスタ
REG0	フラグ
REG1	AX
REG2	BX
REG3	CX
REG4	DX
REG5	SI
REG6	DI
REG7	BP
REG8	DS
REG9	ES

レジスタに値をセットするにはREG文を使います。

```
REG 3,&H0F01
```

この文は，CXレジスタの値を16進数の0F01にセットします。レジスタCHが16進数の0Fに，CLが16進数の01になります。

レジスタの値を読み出すにはREG関数を使います。

```
A%=REG(3)
```

この文は，*A*%をCXレジスタの現在の値にします。

次の例は，画面の表示を行1から行24の方向にスクロールさせます。

```
REG 3,0                    '先頭は行0,桁0
REG 4,&H174F               '最後は行23,桁79
REG 2,&H70                 'カラー7,0
REG 1,&H0701               'BIOSファンクション7:1行スクロール
CALL INTERRUPT &H10        'ビデオ割り込み10h
```

これと同じ働きをするルーチンをアセンブラで書いても，書くのが難しく性能もほとんど向上しません。またこのCALL INTERRUPT形式のほうが変更が容易です。

割り込みを使ってMS-DOSのサービスにアクセスする全手順は，通常非常に簡単です。上級のプログラマには，割り込みはMS-DOSのサービスにアクセスする以外の目的にも使用できます。

割り込みは，温度センサ，遠隔記録装置，タイマのようなデバイスを管理するためによく使用されます。このような割り込みを使うには，まず使用していない割り込みを見つけなければなりません（割り込みの多くはMS-DOSにより使われています。その他の割り込みも，記憶装置やテープドライブなどのために使われている可能性があります）。

Turbo Basicのプログラムの内部で，割り込みベクタがTurbo Assemblerのルーチンを指し示すようにします。Turbo Basicのマニュアルに注意されているように，割り込み処理ルーチンはSSレジスタとSPレジスタの値を退避しなければなりません。その他のレジスタは，変更してもかまいません。割り込み処理ルーチンの終わりでは，**IRET**命令によりTurbo Basicのプログラムに制御を戻します。

先に説明した手法を使ってルーチンのアドレスを決定し，そのアドレスを割り込みベクタのアドレスに入れることができます。ただし割り込みルーチンは，上位メモリに置くか，CALL ABSOLUTEルーチンと同じようにBLOADでロードするのがよい方法です。

＄INLINEコマンドでBasicのプログラムの中に組み込まれた割り込みルーチンは，メモリのどこにロードされているか探し出す必要があります。BLOADにより整数配列に読み込まれるルーチンは，いつも同じアドレスに読み込まれるとは限りませんが，そのアドレスを容易に知ることができます。ただし割り込み処理ルーチンをこのような整数配列に読み込むには，ルーチンの全コードが完全に再配置可能でなければなりません。

そのために割り込みルーチンは，通常上位メモリの決まったアドレスに置きます。この方法を使うときは，Turbo Assemblerのソースコードの中で，**ORG**コマンドにより先頭アドレスを指定する必要があります。

例題プログラム

```
FILLIT2$= CHR$(&HFC) + CHR$(&HF3) + CHR$(&HAB) + CHR$(&HCB)
'                cld          rep          stosw          ret
DIM a%(100)                                  '要素がすべてゼロの整数配列
WHERE%=VARPTR(FILLIT2$)                      '文字列の長さが格納されているアドレス
WHERE%=WHERE%+2                              '文字列のアドレスが格納されているアドレス
CLS:PRINT PEEK(WHERE%),PEEK(WHERE%+1)
HERE%=PEEK(WHERE%)+256*PEEK(WHERE%+1)        '文字列のアドレス
DEF SEG                                      'ここでは必らずしも必要ではないが,
                                             '入れるのが良い習慣

WHERE%=PEEK(0)+256*PEEK(1)
DEF SEG=WHERE%                               '文字列セグメントはデフォルトDS内の
                                             '最初のワード

REGES%=VARSEG(a%(0))
REGSI%=VARPTR(a%(0))
REG 1,5%                                     '配列にフィルする値をAXにロード
REG 3,101%                                   '値をフィルする要素の数(0から100までの
                                             '101個)をCXにロード
REG 9,REGES%                                 '値をフィルする配列のセグメントをESにロード
REG 6,REGSI%                                 '配列の第1の要素のオフセットをSIにロード
CALL ABSOLUTE HERE%                          '配列を5でフィルする
DEF SEG
FOR i%=0 TO 100:PRINT a%(i%);:NEXT i%
PRINT
PRINT REG(1),REG(3),REG(9),REG(6):STOP

CALL FILLIT(a%(0),-1%,101%)                  '配列を-1でフィルする
FOR i%=0 TO 100:PRINT a%(i%);:NEXT i%
PRINT
END

SUB FILLIT INLINE
$INLINE &H55,&H8B,&HEC,&HC4,&H7E
$INLINE &HE,&H26,&H8B,&HD,&HC4
$INLINE &H7E,&HA,&H26,&H8B,&H5
$INLINE &HC4,&H7E,&H6,&HFC,&HF3
$INLINE &HAB,&H5D
END SUB
```

```
; CALL ABSOLUTEルーチンを読み込むためのルーチン(任意の個数の任意の値の要素を
; 整数配列に読み込む). 呼び出し手順は次のとおり.
; REG 1,FILLVALUE%                'AXにはフィルする値がセットされている
; REG 3,FILLCOUNT%                'CXにはフィルする要素の数がセットされている
; REG 9,VARSEG(ARREY(0))          'ESには配列のセグメントがセットされている
; REG 6,VARPTR(ARRAY(0))          'DIには配列の第1の要素のオフセットがセットさ
;                                 'れている
; CALL ABSOLUTE FILLIT2
; FILLIT2はCALL ABSOLUTEルーチンのアドレス.
; DEF SEGは, CALL ABSOLUTEの前にFILLIT2のセグメントアドレスをセットする.

PROGRAM SEGMENT
START   PROC FAR                  ; farリターン型にする
    ASSUME cs:PROGRAM
    push bp                       ; BPを退避
    cld                           ; ディレクションフラグをゼロにする
    rep                           ; CXが0になるまで次の命令を繰り返す
    stosw                         ; AXを[ES:DI]に格納してDIを2インクリメント
    pop bp                        ; BPを回復
    ret                           ; セグメント間リターン(farリターン)
START   ENDP
PROGRAM ENDS                      ; セグメントの終わり
    END

; 任意の個数の任意の値の要素を整数配列に読み込むためのルーチン
; 呼び出し手順: CALL FILLIT(ARRAY(0),FILLVALUE,NUMTIMS)

        ORG 100h
PROGRAM SEGMENT
    ASSUME cs:PROGRAM
    push bp                       ; BPを退避
    mov  bp,sp                    ; SPをBPに転送
    les  di,[bp+0eh]              ; フィルする要素の数のオフセットアドレスを取得
    mov  cx,es:[di]               ; 要素数をCXにロード
    les  di,[bp+0ah]              ; フィルする値のオフセットアドレスを得る
    mov  ax,es:[di]               ; フィルする値をAXにロード
    les  di,[bp+6]                ; フィルする配列のオフセットアドレスを取得
    cld                           ; 方向フラグをゼロにする
    rep                           ; CXが0になるまで次の命令を繰り返す
    stosw                         ; AXを[ES:DI]に格納してDIを2インクリメント
    pop bp                        ; BPを回復
PROGRAM ENDS                      ; セグメントの終わり
    END
```

第9章

Turbo AssemblerとTurbo Prologのインターフェース

Turbo Prologは，画面のウインドウ管理からB＋ツリーデータベースの管理までの多様なハイレベルの機能を提供する多数の述語が用意されています。Turbo Prologのプログラムに Turbo Assemblerのプログラムを組み合わせると，ロウレベルのプログラミング機能を補強することができます。

この章では，まずTurbo AssemblerとTurbo Prologとのインターフェースについて説明します。次にTurbo Prologにインターフェースする簡単ないくつかのアセンブリルーチンの例を示します。最後にアセンブリコードからTurbo Prologの述語を呼び出す方法，Turbo Prologのライブラリコールの使い方，複合構造を渡す方法について説明します。

注意：Turbo Prologはバージョン1.0およびそれ以降のバージョンです。

外部述語の宣言

Turbo Prologでは，**global predicates**宣言を使って他の言語とのインターフェースをとることができます。述語の宣言には，そのグローバル述語がどの言語で実現されているかをTurbo Prologに知らせるために，言語の指定を付けます。

```
global predicates
  add(integer,integer,integer) - (i,i,o),(i,i,i) language asm
  scanner(string,token) - (i,o) language Pascal
```

インターフェースされる言語を明示することにより，アクチベーションレコードおよびパラメータの形式，呼び出しおよびリターンの規則，セグメントの定義，リンク，初期化の問題が簡単になります。

呼び出し規則とパラメータ

8086ファミリのプロセッサでは，サブルーチンを呼び出すのに，nearコールとfarコールのどちらを使うか選択できます。しかしTurbo Prologはラージメモリモデルのプログラムを作成しますので，サブルーチンの呼び出しおよびサブルーチンからのリターンは，すべてfarでなければなりません。

Turbo Prologは，C，Pascal，アセンブラの呼び出し規則をサポートします。Cの呼び出し規則を使って書かれたルーチンを呼び出すときは，パラメータは逆順にプッシュされます。そしてそのルーチンからリターンした後でスタックポインタが自動的に調節されます。他の言語の呼び出し規則を使って書かれたルーチンにインタフェースするときは，パラメータは正順にプッシュされます。スタックからパラメータを取り除くのは，呼び出されたルーチンの責任です。

8086ファミリ用の多くの言語コンパイラでは，16ビットのポインタを使うか32ビットのポインタを使うか選択できます。16ビットのポインタはデフォルトセグメント内のアドレスを指し示します。Turbo Prologは，メモリのどこにアクセスする場合にもいつでも32ビットのポインタを使用します。

Turbo Prologのデータ型は次のとおりです。

integer	2バイト
real	8バイト（IEEEフォーマット）
char	1バイト（スタックにプッシュされるときは2バイト）
string	ヌルで終わる文字列を指す4バイトのダブルワードポインタ
symbol	ヌルで終わる文字列を指す4バイトのダブルワードポインタ
compound	レコードを指す4バイトのダブルワードポインタ

出力パラメータは，戻り値に割り当てられているアドレスを指す32ビットのポインタとしてプッシュされます。入力パラメータは，値が直接プッシュされます。パラメータのサイズは，データ型により異なります。

名前

Turbo Prologでは，同じ述語が，複数のタイプバリアント（異形）やフロー バリアン

トを持つことができます。各タイプバリアントとフローバリアントは，名前の異なるそれ自身のプロシージャを持たなければなりません。Turbo Prologは，各プロシージに0から始まる番号を付け，同一の述語の名前にその番号を付加したものを各プロシージャの名前にします。

たとえば，次のような宣言

```
global predicates
  add(integer,integer,integer)-(i,i,o),(i,i,i) language asm
```

があるとき，フローパターンが（i, i, o）の第1のバリアントには***add_0***という名前が，フローパターンが（i, i, i）の第2のバリアントには***add_1***という名前が付けられます。

またグローバル述語の宣言の後に***"as パブリック名"***部を付け加えることにより，パブリック名を宣言することもできます。次の例では，グローバル述語***pred***は，識別名***pred_0***ではなく***my_pred***により解決されます。

```
global predicates
  pred(integer,integer)-(i,o) language asm as "my_pred"
```

この方法は，フローパターンが1つしかない述語に名前を付けるときに便利です。複数のフローパターンがあるときは，各バリアントに名前をつけなければなりません。例として***add***述語を使うと，述語の定義はたとえば次のようになります。

```
global predicates
  add(integer,integer,integer)-(i,i,o) language asm as "doadd"
  add(integer,integer,integer)-(i,i,i) language asm as "add_check"
```

この例では，フローパターンが（i, i, o）の第1のバリアントには***doadd***という名前を付け，フローパターンが（i, i, i）の第2のバリアントには***add_check***という名前をつけています。この命名法では，バリアントを別々に宣言しなければならないことに注意してください。

アセンブリ言語の述語を書く

書くのが最も簡単な述語は，たぶん入力フローパターンだけの述語です。いまTurbo Prologのカレントウィンドウ内の表示を，水平にスクロールさせたいとしましょう。そして画面のある領域を1桁左にスクロールする述語***scroll_left***を作成しましょう。例題のSCROLLH.PROの中では***scroll_left***は，4つの整数引数と1つのフローパターンを取ります。

Turbo PrologのモジュールSCROLLH.PROは，***scroll_left***述語の**global predicates**宣言を含みます。***scroll_left***述語は，アセンブリ述語として定義されています。

```
/* SCROLLH.PRO */

global predicates
  scroll_left(integer,integer,integer,integer) - (i,i,i,i) language asm

predicates
  scrollh
clauses
  scrollh :-
    makewindow(_,_,_,_,Row,Col,Nrows,Ncols),
    scroll_left(Row,Col,Nrows,Ncols),
    readchar(C),
    char_int(C,CI),
    not(CI = 27),
    scrollh.

goal
  makewindow(1,7,7," A SCROLLING MESSAGE ",10,20,4,60),
  write("This message will scroll across the window"),nl,
  write("Look at it go!"),
  readchar(_),
  scrollh,
readchar(_).
```

次のアセンブリ言語のソースは，***scroll_left***述語の実現です。このプロシージャに付けられた名前が*SCROLL_LEFT_0*であることに注意してください。
この名前は，この章で先に説明した命名規則に合っています。

```
; SCROL.ASM
;
    name      scrol
;
; scroll_left(integer,integer,integer,integer) - (i,i,i,i) language asm
;
SCROL_TEXT     SEGMENT   BYTE PUBLIC 'CODE'
    ASSUME     CS:SCROL_TEXT
```

```
PUBLIC SCROLL_LEFT_0

SCROLL_LEFT_0   PROC FAR
;
; パラメータ
arg  NCOLS:WORD, NROWS:WORD, COL:WORD, ROW:WORD = ARGLEN
;
; ローカル変数
local      SSEG :WORD = LSIZE

     push  bp
     mov   bp,sp
     sub   sp,lsize              ; ローカル変数のための領域を確保
     push  si
     push  di

     mov   SSEG, 0B800h

     sub   NCOLS, 3              ; NCOLS = NCOLS - 3

     mov   ax, ROW               ; DEST = ROW*160 + (COL+1)*2
     mov   dx,160
     mul   dx
     mov   dx, COL
     inc   dx                    ; +1
     shl   dx,1
     add   dx,ax

     push  ds
     push  es

     mov   bx ,NROWS             ; BXをカウンタとして使ってNROWS回ループ
     dec   bx                    ; NROWS = NROWS - 2
     dec   bx

Top: cmp   bx ,0
     je    Done

     add   dx, 160               ; dest = dest + 160

     mov   ax,NCOLS              ; lastchar = dest + nc*2
     shl   ax,1
     add   ax,dx
     push  ax                    ; lastcharのオフセットをスタックにプッシュ

     mov   ax,SSEG               ; 画面セグメントをES,DSにロード
     mov   es,ax
     mov   ds,ax

     mov   di,dx                 ; 転送のためにSIとDIをセットアップ
     mov   si,di                 ; ソースはDESTの上の2バイト
     add   si,2

     mov   ax,[di]               ; 桁0のキャラクタをAXに退避
```

```
        mov     cx,NCOLS                ; NCOLSワード転送
        cld
        rep     movsw

        pop     di                      ; lastcharのオフセットをDIにポップ
        mov     [di],ax                 ; AXのキャラクタを最後の桁に置く

        dec     bx
        jmp     TOP
Done:pop     es
        pop     ds
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret     ARGLEN
SCROLL_LEFT_0   ENDP
SCROL_TEXT      ENDS
        END
```

SCROLLH.PROとSCROL.ASMから実行可能ファイルを作成するには，まずTurbo PrologによりTurbo Prologファイルをコンパイルします。Turbo Prologでモジュールをコンパイルすると，.OBJファイルと.SYMファイルが作成されます。次にTurbo AssemblerでSCROL.ASMをアセンブルして.OBJファイルを作成します。さらに次のTLINKコマンドラインで，2つのモジュールを互いにリンクします。

```
TLINK init scrollh scrol scrollh.sym,scroll,,prolog
```

これにより，SCROLL.EXEという名前の実行可能ファイルが生成されます。

double述語の実現

アセンブリ言語ルーチンが，次の節により呼び出されるものとしましょう。

```
double(MyInVar,MyOutVar)
```

*MyInVar*は呼び出しの前にある整数値に限定され，*MyOutVar*は呼び出し後その整数値の2倍の整数値に限定されます。

doubleがアクチベートされているとき（呼び出されているとき）にスタックトップに置かれるアクチベーションレコードは，図9.1に示すような形式になります。

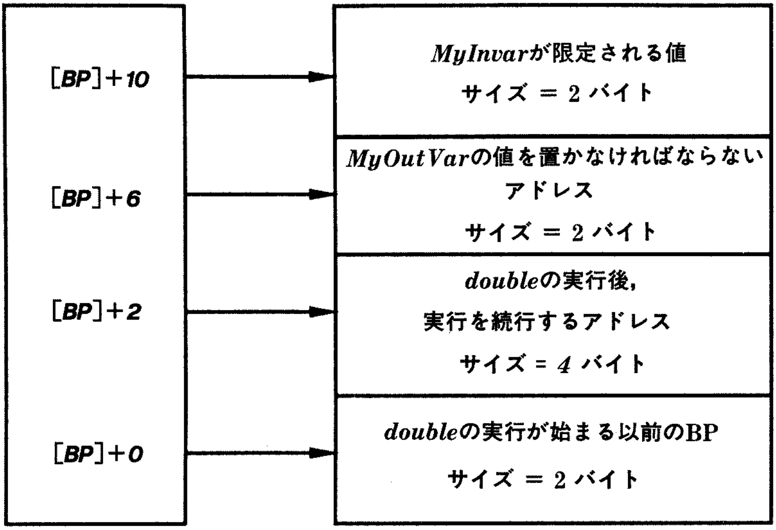


図9.1 doubleのアクチベーションレコード

次のアセンブリ言語の関数は，***double***述語を実現するプログラムです。

```
; MYASM.ASM
;
A_PORG     SEGMENT BYTE
           ASSUME  CS:a_prog
           PUBLIC  double_0
double_0   PROC    FAR
           push    bp
           mov     bp,sp

           mov     ax,[bp+6]                ; MyInVarが限定される値を取得する
           add     ax,ax                    ; その値を2倍する
           lds     si,DWORD PTR [bp]+10
           mov     [si],ax                  ; MyOutVarが限定される値を
                                            ; [ES:SI]に格納
           pop     bp
           mov     sp,bp
           ret     6
double_0   ENDP
A_PROCG    ENDS
```

doubleの呼び出しを含むTurbo Prologのプログラムには，次の**global predicates**宣言がなければなりません。

```
global predicates
  double(integer,integer) - (i,o) language asm
```

それ以外の点については他のプログラムと同じです。

次のプログラムはこの***double***を使っています。

```
/* MYPROG.PRO */

global predicates
  double(integer,integer) - (i,o) language asm

goal
    write("Enter an integer "),
    readint(I),
    double(I,Y),
    write(I," doubled is ",Y).
```

このアセンブリ言語のプログラムモジュールがアセンブルされてファイルMYASM.OBJが作成され，それを呼び出すTurbo PrologのオブジェクトモジュールがMYPROLOG.OBJのとき，次のコマンドライン

```
TLINK init myprolog myasm myprolog.sym,double,,prolog
```

によりこの2つのモジュールをリンクし，実行可能なスタンダロンプログラムDOUBLE.EXEを生成することができます（PROLOG.LIBの中のTurbo Prologライブラリを使います)。MYPROG.SYMが，TLINKのコマンドラインの，最初のカンマの前の最後のファイル名として指定されていることは重要です。

一般にアクチベーションレコードの形式は，呼び出し側のTurbo Prologの述語の中のパラメータの数と，それらのパラメータに対応するドメインの型により異なります。たとえば，***integer***ドメインに属するパラメータ*Val1, Val2, Sum*を持つ次の述語

```
add(Val1,Val2,Sum)
```

を定義したいときは，アクティベーションレコードは次の図9.2のようになるでしょう。

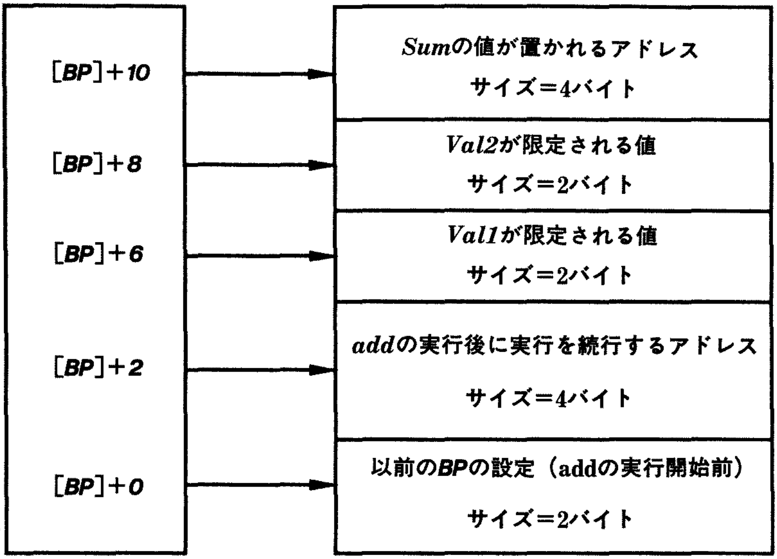


図9.2　addのアクチベーションレコード

各パラメータは，対応するバイト数を占めることに注意してください。出力パラメータはいつでも4バイトです（セグメントアドレスとオフセットのために使用されます）。入力パラメータのサイズは，実際にスタックにプッシュされる値により決まります。したがって対応するドメインの型によりサイズが異なります。

*Val1*と*Val2*は，***integer***ドメインに属し，(i)フローパターンで使用されます。両方とも2バイトを占めます。*Sum*は，(o) フローパターンで使用され，4バイトを占めます。

Turbo Prologのコンパラの内部では，外部述語の呼び出しは次のような形になることに注意してください。

```
mov  ax,SEGMENT data
mov  ds,ax
call FAR PTR external_predicate_implementation
```

したがって外部述語のプロシージャが実行されている間にアクセスされるデータセグメントは，**DATA**という名前のセグメントです。

複数のフローパターンを持つ述語の実現

複数のフローパターンを持つ述語を実現するときは，Turbo Prologの命名規則にしたがってアセンブリ言語の関数に名前を付けるように注意してください。たとえば，複数のフローパターンを持つ述語***add***を実現したいとしましょう。述語***add***は，***add***が呼び出されて3つの引数のうちのどれか2つが限定されるとき，式$X+Y=Z$の中の足りない値を求めます。

次のTurbo PrologのプログラムADDPRO.PROは，グローバルなアセンブリ言語の述語***add***を宣言します。***add***は3つのフローパターン（i, i, o），（i, o, i），（o, i, i）を持つことに注意してください。

```
/* ADDPRO.PRO */

global predicates
  add(integer,integer,interger) - (i,i,o),(i,o,i),(o,i,i) language asm

goal
   add(2,3,X), write("2 + 3 = ",X),nl,
   add(2,Y,5), write("5 - 2 = ",Y),nl,
   add(Z,3,5), write("5 - 3 = ",Z),nl.
```

次のアセンブリ言語のプログラムADD.ASMは，***add***を実現するためのコードを含んでいます。**ADD_0**は (i,i,o) フローパターンに，**ADD_1**は (i,o,i) フローパターンに，**ADD_2**は（o,i,i）フロー パターンに対応します。

```
        name      add

ADD_TEXT          SEGMENT   BYTE PUBLIC 'CODE'
        ASSUME    CS:ADD_TEXT
        PUBLIC    ADD_0                           ; (i,i,o)フローパターン
ADD_0   PROC      FAR
        arg       Z:dWORD, Y:WORD, X:WORD = ARGLEN1
        push      bp
        mov       bp,sp
        mov       ax,X
        add       ax,Y
        les       bx,Z
        mov       WORD PTR ES:[BX],ax
        pop       bp
        ret       ARGLEN1
ADD_0   ENDP
```

```
        PUBLIC      ADD_1                           ; (i,o,i)フローパターン
ADD_1   PROC        FAR
arg     Z:WORD, Y:dWORD, X:WORD = ARGLEN2
        push        bp
        mov         bp,sp
        mov         ax,Z
        sub         ax,X
        les         bx,Y
        mov         WORD PTR ES:[BX],AX
        pop         bp
        ret         ARGLEN2
ADD_1   ENDP

        PUBLIC      ADD_2                           ; (o,i,i)フローパターン
ADD_2   PROC        FAR
arg     Z:WORD, Y:WORD, X:dWORD = ARGLEN1
        push        bp
        mov         bp,sp
        mov         ax,Z
        sub         ax,Y
        les         bx,X
        mov         WORD PTR ES:[BX],AX
        pop         bp
        ret         ARGLEN3
ADD_2   ENDP
ADD_TEXT        ENDS
        END
```

ADDPRO.PROとADD.ASMを.OBJファイルに変換した後，次のコマンドラインにより.EXEファイルを作成することができます。

```
TLINK init addpro add addpro.sym,addpro,,prolog
```

アセンブリ関数からTurbo Prologの述語を呼び出す

Turbo Prologの述語からアセンブリ言語の関数を呼び出す方法を説明しましたので，次にその反対にアセンブリ関数からTurbo Prologの述語を呼び出す方法を説明します。

述語がグローバル述語として宣言されているときは，その述語のバリアントは他のすべてのモジュールから呼び出すことができるグローバル関数になります。名前のつけ方と呼び出し方は，アセンブリ言語で定義された述語の場合と同じです。

次のTurbo Prologのモジュールは, 2つのグローバル述語***popmessage***と***from_asm***を定義します。***popmessage***はC言語の述語として宣言され，***from_asm***はアセンブリ言語の述語として宣言されています。

SHOWMESSを作成するには, まずTurbo Prologの開発環境でSHOWMESS.PROを.OBJにコンパイルします。次にFROM_ASM.ASMをアセンブルします。

```
tasm from_asm
```

さらに2つのモジュールをリンクします。

```
TLINK init showmess from_asm showmess.sym,showmess,,prolog
```

SHOWMESSは次のようになります。

```
/* SHOWMESS.PRO */

global predicates
  popmessage(string) - (i) language c     /* アセンブリ言語のプロシージャから
                                             呼び出される述語               */
  from_asm - language asm                 /* アセンブリ言語のプロシージャ */

clauses
  popmessage(S) :-                        /* popmessage_0という名前のCの
                                             関数として呼び出すことができる */
    str_len(S, L),
    LL = L + 4,
    makewindow(13,7,7,"",10,10,3,LL),
    write(S),
    readchar(_),
    removewindow.

goal
    from_asm.                             /* 外部 */
```

次に示すアセンブリコードは，***from_asm***を実現し，***popmessage***を呼び出します。

```
        EXTRN   PopMessage_0:FAR
DGROUP  GROUP   _DATA
        ASSUME  CS:SENDMESS_TEXT,DS:DGROUP
_DATA   SEGMENT WORD PUBLIC 'DATA'
messl   DB      "Report: Condition Red",0
_DATA   ENDS

SENDMESS_TEXT   SEGMENT     BYTE PUBLIC 'CODE'
        PUBLIC  FROM_ASM_0
FROM_ASM_0      PROC    FAR
        push    ds
        mov     ax,OFFSET DGROUP:messl
        push    ax
        call    FAR PTR PopMessage_0
        pop     cx
        pop     cx
        ret
FRAM_ASM_0      ENDP
SENDMESS_TEXT   ENDS
        END
```

次にアセンブリ言語のハイレベルの拡張機能を使って，上と同じ実行可能なプログラムを作成しましょう。まずTurbo Prologの開発環境でSHOWNEW.PROを .OBJにコンパイルます。次に

```
tasm /jmasm51 /jquirks from_new
```

によりFRPOM_ASM.ASMをアセンブルし，さらに次のコマンドでリンクします。

```
TLINK init shownew from_new shownew.sym,show2,,prolog
```

SHOWNEW.PROは次のようになります。

```
/* SHOWNEW.PRO */

global predicates
  popmessage(string) - (i) language c      /* アセンブリ言語のプロシージャから
                                              呼び出される述語                 */
  from_asm - language c as "_from_asm"     /* アセンブリ言語のプロシージャのパブ
                                              パブリック名を定義する           */
clauses
  popmessage(S) :-
    str_len(S,L),
    LL=L+4,
    makewindow(13,7,7,"window",10,10,3,LL),
    write(S),
    readchar(_),
    removewindow.

goal
    from_asm.                               /* アセンブリ言語のプロシージャを
                                              呼び出す                       */
```

次のアセンブリコードは，***from_asm***を実現し，先の例と同じように***popmessage***を呼び出します。

```
; FROM_NEW.ASM

      extrn   PopMessage_0:far
      .model  large,c
      .code

FROM_ASM      proc
      push    ds
      mov     ax, OFFSET DGROUP:mess1
      push    ax
      call    FAR PTR PopMessage_0
      pop     cx
      pop     cx
      ret
FROM_ASM     ENDP

      .data
mess1 DB     "report: Condition Red",0
      END
```

リストとファンクタ

このセクションでは，アセンブリ言語の述語にリストとファンクタを渡すために使用される方法を説明します。先に説明したようにTurbo Prologは，複合項は直接渡しません。そのかわりに構造体を指す4バイトのポインタを渡します。

リストとファンクタの両方に使われるレコード構造体は，簡単で直接的です。いま次のようなTurbo Prologのドメインがあるとしましょう。

```
domains
    ilist = integer*
    ifunc = int(integer)
```

ilistドメインに対するリストノード構造体は，次のようになります。

```
STRUCT ILIST
  NodeType  DB ?
  Value     DW ?
  NextNode  DD ?
ENDS
```

この構造体からわかるように，リストノードは3つの部分からなります。

■ノードタイプ（1バイト）

■ノード値（ノードのタイプに依る）
■次のノードを指すポインタ（4バイト）

ノードタイプは，2つの意味のある値を含むことができます。*Value1*はそのノードがリストのノードであることを意味し，*Value2*はそのノードがリストの最後のノードであることを示します（リストの最後のノードには，他に意味のある情報を含みません)。ノード値は，Turbo Prologのどのドメインの値でもかまいせん。

ifuncファンクタの対応する構造体は次のようになります。

```
STRUC IFUNC
  FuncType   DB   ?
  Value      DW   ?
ENDS
```

ファンクタ構造体は，ファンクタタイプとファンクタレコードの2つの部分からなります。ファンクタタイプは，バリアントファンクタのリストの中のそのファンクタバリアントの位置を表わす整数です。最初のバリアントのタイプは1, 2番目のバリアントはタイプ2, 以下同様になります。

次のTurbo PrologのモジュールとTurbo Assemblerのモジュールの中で, Turbo Prologにファンクタを返す述語を実現しています。

Turbo Prologのモジュールは次のとおりです。

```
/* FUNC.PRO */

domains
  ifunc = int(integer)

global predicates
  makeifunc(integer,ifunc) - (i,o) language c

goal
  makeifunc(4,H),
  write(H).
```

Turbo Assemblerのモジュールは次のとおりです。

```
; IFUNC.ASM
;
    EXTERN     _alloc_gstack:FAR                ; _alloc_gstackはメモリブロック
                                                ; を指すポインタを返す.
STRUC   IFUNC
  FuncType   DB ?
  Value      DW ?
ENDS

IFUNC_TEXT   SEGMENT   WORD PUBLIC 'CODE'
     ASSUME  CS:IFUNC_TEXT
```

```
    PUBLIC   Makeifunc_0
Makeifunc_0  PROC FAR
    arg      __inval:WORD,__outp:dWORD
    push     bp
    mov      bp,sp
    mov      ax,3                          ; 3バイト確保
    push     ax
    call     FAR PTR _alloc_gstack
    pop      cx
    les      bx,__outp
    mov      [WORD PTR ES:BX+2],DX
    mov      [WORD PTR ES:BX],AX
    mov      ax,__inval
    les      bx,__outp
    les      bx,[DWORD PTR ES:BX]
    mov      [(IFUNC PTR ES:BX).VALUE],AX    ; 値=__inval
    mov      [(IFUNC PTR ES:BX).FUNCTYPE],1  ; タイプ=1
    pop      bp
    ret
Makeifunc_0  ENDP
IFUNC_TEXT   ENDS
    END
```

この例では，***ifunc***のために1つのファンクタタイプだけを使っています。もう1つのファンクタを宣言したいときは，次のようにします。

```
myfunc = int(integer); char(char); r(real); d2(integer,real)
```

この場合には，構造体はいくらか複雑になりますが，やはり2つの部分を持ちます。ただし第2の部分は，***myfunc***のすべてのバリアントを定義するために必要なデータ構造の共用体になるでしょう。次の構造体は，Turbo Assemblerによる***myfunc***の実現の1例です。

```
STRUCT MyFunc
       FuncType  DB ?
       UNION
          STRUC
             _int   DW ?
          ENDS
          STRUC
              _char DB ?
          ENDS
          STRUC
              _real DQ ?
          ENDS
          STRUC
              v1    DW ?
              v2    DQ ?
          ENDS
       ENDS
ENDS
```

ファンクタバリアントのとり得るタイプには，次のものがあります。

int(integer)	1
char(char)	2
r(real)	3
d2(integer, real)	4

リストとファンクタについては，先の***ilist***の**domains**宣言を見てください。なぜ有効なノードタイプ1および2があるのでしょうか? なぜなら，Turbo Prologは***ilist***を構造体と見なすからです。この構造体は，次のように簡単に宣言できるでしょう。

```
ilist = listnode(integer,listnode); end_of_list.
```

複合項を渡すときは，構造体を指すポインタを渡すことを忘れないでください。入力のフローパターンリストまたはファンクタは，参照呼び（オフセットのみのポインタ，セグメントはDSを使用）で渡されます。出力のフローパターンリストまたはファンクタは，構造体を指すポインタのアドレスで渡されます(Turbo Prologは，返される構造体を指すポインタのアドレスを渡します)。Turbo Prologに返すすべての構造体は，Turbo Prologのメモリアロケーション関数で確保されたメモリを使うべきです(*Turbo Prolog User's Guide*および*Turbo Prolog Reference Guide*を参照してください)。

第10章

Turbo Assemblerの応用プログラミング

前の章までは，アセンブラプログラミングの基本といくつかの応用例を説明してきました。この章では，Turbo Assemblerのさらに進んだ機能について学習します。

この章では，今まではほんの概要を説明したアセンブラプログラミングの要素（セグメントオーバライドプレフィクス，マクロ命令，セグメント疑似命令，複数のコードセグメントとデータセグメントの作り方など）をさらに詳しく説明します。ローカルラベル，自動ジャンプサイズ決定，前方参照，データ構造疑似命令などの新しい便利な機能についても解説します。

このマニュアルでは，アセンブリ言語プログラミング一般について説明するのはこの章が最後です。これまではアセンブラプログラミングの基本について紹介し，多くの専門的な話題にも触れてきましたが，けしてアセンブリ言語プログラミングのすべてを説明したわけではありません。このマニュアルの巻末でアセンブリ言語プログラミングに関する本を何冊か紹介します。これらの本の中から選んで，または役に立つ本を入手してお読みになり，アセンブリ言語の学習を続けてください。

セグメントオーバライドプレフィクス

メモリオペランドは，通常DSセグメントレジスタの指すセグメントのメモリアドレスを指定します。次に例を示します。

```
.
.
.
mov   bx,10h
mov   si,5
mov   ax,[bx+si+1]
.
.
.
```

この例は，DSの指すセグメント内のオフセット16hに格納されているワードをAXにロードします。つまりメモリアドレスDS:0016の内容をAXにロードします。

DSの指すセグメントからデータをロードをする規則にも例外があります。たとえばストリング命令**STOS**と**MOVS**は，ESの指すセグメントをデスティネーションオペランドとします。ストリング命令**SCAS**と**CMPS**は，ESの指すセグメントをソースオペランドとします（**CMPS**のソースオペランドの一方はデータセグメント内にあり，他方はエクストラセグメント内にあります）。

この他の例外としては，BPを伴う任意のメモリオペランドはSSの指すセグメントをアクセスすることがあります。次に例を示します。

```
.
.
.
mov   bp,1000h
mov   al,[bp+6]
.
.
.
```

この例はメモリアドレスSS:1006の内容をALにロードします。

では，CSセグメント内のアドレスをメモリオペランドとしてアクセスするにはどうすればよいのでしょうか？　これは特にマルチセグメントプログラム内のジャンプテーブルに使うと便利です。またBXでスタック上のアドレスに，BPでDS内のアドレスに，ストリン

グ命令以外の命令でES内のアドレスに，それぞれアクセスすることは可能でしょうか？

答はイエスです。**セグメントオーバライドプレフィクス**を使えば，多くの命令で指定したセグメントをアクセスさせることができます。次に例を示します。

```
.
.
.
mov   bX,100h
mov   cl,ss:[bx+10h]
.
.
.
```

この例はスタックセグメント内のオフセット110hの内容をCLにロードします。

```
.
.
.
mov   bp,200h
mov   si,cs:[bp+1]
.
.
.
```

この例はコードセグメント内のオフセット201hの内容をSIにロードします。

基本的には，デフォルトセグメント以外のセグメントにアクセスするためには，特定の命令のメモリオペランドの前にセグメントオーバライドプレフィクス（CS:, DS:, ES:, SS:）を付ければよいのです。

セグメントオーバライドプレフィクスは，命令のメモリオペランドのプレフィクスとして付けられるため，厳密には"プレフィクス"とは言えません。セグメントオーバライドプレフィクスは，第5章で説明したREPプレフィクスと同じように，実質的には命令プレフィクスバイトであり，後続命令の作用を修正するものです。たとえば次の8086の命令コードは，データセグメント内のオフセット0の内容をALにロードします。

```
A0 00 00      ( ← mov   al,[0] )
```

これにES:セグメントオーバライドプレフィクスの値26hを付けると，次のようになります。

```
26 A0 00 00      ( ← mov   al,es:[0] )
```

この命令は，データセグメントではなく，エクストラセグメント内のオフセット0の内容をALにロードします。

セグメントオーバライドプレフィクスの別の形式

Turbo Assemblerは別の形式のセグメントオーバライドプレフィクスもサポートします。これを使えば，セグメントオーバライドプレフィクスを別の行として記述できます。別の行のセグメントオーバライドは次のようになります。

CS:セグメントオーバライド → **SEGCS**
DS:セグメントオーバライド → **SEGDS**
ES:セグメントオーバライド → **SEGES**
SS:セグメントオーバライド → **SEGSS**

これらのプレフィクスは，次の行のコードをオーバライドするだけで後続のすべての行をオーバライドするわけではありません。次の例はエクストラセグメント内のオフセット999hにDXの値をストアします。

```
.
.
.
mov  si,999h
seges
mov  [si],dx
.
.
.
```

この形式は，**LODSB**などのオペランドのない命令に対してセグメントオーバライドプレフィクスを付けるのに便利です。次の例はSS:SIが指す内容をALにロードします。

```
.
.
.
segss
lodsb
.
.
.
```

セグメントオーバライドプレフィクスが使えない場合

セグメントオーバライドプレフィクスはすべての命令に使えるわけではありません。たとえば，エクストラセグメントにアクセスするストリング命令はオーバライドできません。たとえば，次の命令はES:SIの内容をALにロードします。

```
lods  es:[ByteVar]
```

しかし次の命令はうまく働きません。

```
stos  ds:[ByteVar]
```

エクストラセグメントにアクセスするストリング命令をオーバライドしようとすると，Turbo Assemblerはそれができない旨のメッセージを出力します。しかしセグメントオーバライドプレフィクスとして**SEGCS**などを使った場合は，Turbo Assemblerはどの命令をオーバライドしようとしているかがわからないため，エラーとはなりません。次に例を示します。

```
.
.
.
segds
stosb
.
.
.
```

この例はアセンブルエラーとはなりません。**STOSB**はデータセグメントではなくエクストラセグメントに対して書き込みを行ないます。

セグメントオーバライドプレフィクスはスタックへのアクセスに対しては影響を与えないことに注意してください。スタックへのプッシュとポップは必らずスタックセグメントに対して行なわれます。次に例を示します。

```
.
.
.
segcs
push [bx]
.
.
.
```

この例はセグメントオーバライドプレフィクスを使って，プッシュする値をどのセグメ

ントからフェッチするのかを指定します。その値は常にスタックセグメント内のオフセットSP-2にストアされます。同様に，命令は常にCSの指すセグメントからフェッチされます。

複数のオーバライドプレフィクスを使う命令の実行中に割り込みが発生すると，エラーの原因となります。したがって通常は上記のオーバライドプレフィクスを**REP**プレフィクスといっしょに使わないでください（詳しくは第5章を参照してください）。

複数セグメントにアクセスする

複数のセグメントにアクセスする必要のあるときは，セグメントオーバライドプレフィクスは便利です。セグメントオーバライドプレフィクスは，たとえばスタックセグメントとデータセグメントの双方にストアされたデータをアクセスしなくてはならない場合に必要となります。これはスタックが動的に配置された変数に使われ，データセグメントが静的な変数に使われる場合によくあることです。他の要因としては，プログラムに64Kバイトを超えるデータを使う場合です。この場合は，常に複数セグメントのどれかにアクセスしなければなりません。

セグメントオーバライドプレフィクスが特に便利な例の1つとして，ストリング命令とそれ以外の命令を混ぜて使う必要のある場合があります。たとえば，ある文字列内の20h未満の文字をすべてスペースに変換する場合を考えてみます。以下のプログラムは，セグメントオーバライドプレフィクスを使ってこの処理を効率よく行ないます。

```
        .
        .
        .
        mov   ax,SEG StringToConvert
        mov   es,ax
        mov   di,OFFSET StringToConvert     ; ES:DIは変換する文字列をポイント
        cld                                 ; STOSBではDIをカウントアップする
ConvertLoop:
        mov   al,es:[di]                    ; 次の文字をロード
        and   al,al                         ; 文字列の終端か？
        jz    ConvertLoopDone               ; Yes...終了
        cmp   al,20h                        ; 変換すべき文字か？
        jnb   SaveChar                      ; No... 文字をストア
        mov   al,' '                        ; スペースに変換
SaveChar:
        stosb                               ; この文字をストアして次の文字をポイント
        jmp   ConvertLoop                   ; 次の文字をチェック
ConvertLoopDone:
        stosb                               ; 文字列の終端に0をストア
        .
        .
        .
```

ローカルラベル

ローカルラベル(限られた範囲(スコープ)でのみ使用できるラベル)は, Turbo Assemblerを使う楽しみの1つでもあります。ではなぜローカルラベルが必要なのかを説明します。

ソースモジュール内に似たような機能をもつ2つまたはそれ以上の部分がある場合を例にとります。たとえば, 次のようなプログラムを考えてみます。

```
      .
      .
      .
Sub1 PROC
      sub   ax,ax
IntCountLoop:
      add   ax,[bx]
      inc   bx
      inc   bx
      loop IntCountLoop
      ret
Sub1 ENDP
      .
      .
      .
Sub2 PROC
      sub   ax,ax
      mov   dx,ax
LongCountLoop:
      add   ax,[bx]
      adc   dx,[bx+2]
      add   bx,4
      loop LongCountLoop
      ret
Sub2 ENDP
      .
      .
      .
```

この例の2つの部分が同じような処理を行なう場合, 結果として類似ラベルが含まれることはよくあることです。この例では, *Sub1*と*Sub2*の両方にカウントループ処理の先頭を示す類似ラベルがあります。

プログラム内に含まれるラベルの総数がそれほど多くない場合は, ラベルが全部異なることを簡単に確かめられます。しかし大きなプログラムでは, ラベルが重複しないようにするのは難しくなります。同じような機能のサブルーチンを別に作る場合は, 通常は目的のサブルーチンをブロックコピーして名前を変え, 変更を加えて新しいサブルーチンとして使います。この場合に問題となるのは, あちこちでラベルの名前を変えるのを忘れてしまい, 実行時に新しいサブルーチンから古いサブルーチン内のラベルへジャンプする可能

性があることです。たとえば，*Sub1*をコピー，修正して*Sub2*を作った場合，うっかり次のようにラベルを変更し忘れることがあります。

```
      .
      .
      .
Sub2 PROC
      sub  ax,ax
      mov  dx,ax
LongCountLoop:
      add  ax,[bx]
      adc  dx,[bx+2]
      add  bx,4
      loop IntCountLoop
      ret
Sub2 ENDP
      .
      .
      .
```

これでは，loop命令で*Sub1*の内にジャンプしてしまいます。悲惨な結果になる可能性は十分にあります。

このような場合に必要なのは，1つのサブルーチン内にスコープが限定されたラベルです。これなら他のサブルーチン内のラベルと重複しません。

このようなラベルをローカルラベルと言います。ローカルラベルは，指定のない限り2つの単価記号（@@）で始まります。ローカルラベルは，2つの非ローカルラベルにはさまれた範囲にスコープが限られます(非ローカルラベルとは，ローカルラベルを包括する**PROC**と対応する**ENDP**につけたラベルを言います)。Turbo Assemblerに関する限り，ローカルラベルのスコープは隣の非ローカルラベルを越えることはありません。

LABEL疑似命令でシンボル定義した場合も，新しいローカルシンボルのブロックが開始することはありません。

ローカルラベルを使えば，この節のはじめの例を以下のように変更できます。

```
      .
      .
      .
      LOCALS
Sub1 PROC
      sub  ax,ax
@@CountLoop:
      add  ax,[bx]
      inc  bx
      inc  bx
      loop @@CountLoop
      ret
Sub1 ENDP
      .
      .
```

```
        .
  Sub2 PROC
        sub   ax,ax
        mov   dx,ax
  @@CountLoop:
        add   ax,[bx]
        adc   dx,[bx+2]
        add   bx,4
        loop @@CountLoop
        ret
  Sub2 ENDP
        .
        .
        .
```

こうすれば，一方のサブルーチン内のラベルと他のサブルーチン内の同名のラベルとが混同する心配はありません。したがって，一方のサブルーチンから誤って他のサブルーチン内のラベルへジャンプする心配はありません。

お気づきでしょうが，この例ではローカルラベルを使用する前に**LOCALS**疑似命令を使っています。MASMモードではローカルラベルはデフォルトでは使用できないため，ローカルラベル使用前に**LOCALS**を宣言して使用可能にする必要があります。Idealモードでは，ローカルラベルは常時使用できます。必要に応じて，**NOLOCALS**を使ってローカルラベルを使用できないようにすることもできます。

ローカルラベルは，サブルーチン内にいくつかのショート条件ジャンプがある場合にも便利です。ローカルラベルを使えば，ラベルの名前が重複しないように心配する必要はなくなります。たとえば，値をいくつかテストするためにローカルラベルを使う場合があります。

```
        .
        .
        .
        LOCALS
        cmp   al,'A'
        jnz   @@P1
        jmp   HandleA
  @@P1:
        cmp   al,'B'
        jnz   @@P2
        jmp   HandleB
  @@P2:
        cmp   al,'C'
        jnz   @@P3
        jmp   HandleC
  @@P3:
        .
        .
        .
```

ローカルラベルを使えば，*P1*などのラベルがプログラム内の別の場所で使われていない

かを心配する必要はありません。

非ローカルラベルはどんなものでも，ローカルラベルのスコープを限定することに注意してください。次のプログラムは正しくアセンブルできません。

```
      .
      .
      .
Sub1 PROC NEAR
      .
      .
      .
      LOCALS
@@CountLoop:
      add  ax,[bx]
      jnz  NotZero
      inc  dx
Notzero:
      inc  bx
      inc  bx
      loop @@CountLoop
      .
      .
      .
```

この例では，非ローカルラベル*NotZero*がローカルラベル@@*CountLoop*と@@*CountLoop*への**LOOP**命令の間にあることが問題です。ローカル変数のスコープは隣の非ローカルラベルまでに限定されるため，Turbo Assemblerが**LOOP**命令をアセンブルするとき，ローカルラベル@@*CountLoop*は見つかりません。

ローカルシンボルのプレフィクスとして，2つの単価記号（@@）から（シンボル名の最初に使える）任意の他の2文字に変更することができます。それには，**LOCALS**疑似命令の引数に新しいプレフィクス文字を指定します。次に例を示します。

```
LOCALS __
```

この例はローカルシンボルプレフィクスとして，2つの下線記号（__）を指定します。デフォルトのローカルシンボルプレフィクス（@@）を使ったシンボルがすでに使われているモジュールの中でローカルシンボルを使いたい場合に便利です。

このようにローカルシンボルプレフィクスを変更した場合は，引数のない**LOCALS**疑似命令を使ったときと同じように，自動的にローカルシンボルが使用可能になります。また，この後に**NOLOCALS**疑似命令でローカルシンボルを使用不能に変更しても，以前指定したプレフィクス文字は記憶されています。したがって**LOCALS**を単独に（引数を付けないで）指定すれば，以前指定したプレフィクスの付いたローカルシンボルが再び使用可能になります。

自動ジャンプサイズ決定

8086が設計された当時は,条件ジャンプ命令は1バイトで表わせるオフセットのみをサポートするように決められました。つまり，条件ジャンプにおいてはジャンプ命令自身から前後に約128バイトの範囲内にしかジャンプすることができません。

もちろん今日でもこれらの条件ジャンプはそのままサポートされていますが，これは良い面と悪い面があります。8086の条件ジャンプ命令からはコンパクトなコードが生成されることもありますが（条件ジャンプ命令は2バイト長)，効率の悪いコードが生成されることもあります。条件ジャンプのジャンプ先が1バイトの変位で表わせない場合には，次のように5バイトの命令シーケンスが必要となります。

```
        .
        .
        .
        jnz   NotZero
        jmp   IsZero
NotZero:
        .
        .
        .
```

さらに悪いことに，与えられた条件ジャンプが目的のラベルに到達するか否かを前もって知る方法はありません。したがって，ラベルに直接ジャンプせざるを得ません。これによってアセンブルエラーの危険を冒し，無条件ジャンプと条件ジャンプのペアでコードを生成するため3バイトを浪費し，処理速度を低下させることになります。さらに困るのは，ループ内に命令を追加すると**Relative jump out of range**のエラーが頻繁に発生することです。

Turbo Assemblerは8086の条件ジャンプの問題をすべて解決しているわけではありませんが，**JUMPS**疑似命令によってかなりの程度解決しています。この**JUMP**疑似命令を指定すれば，通常の条件ジャンプに対して無条件ジャンプと条件ジャンプのペアを生成して，目的のラベルへジャンプさせることができます。

次に自動ジャンプサイズ決定のプロセスを説明します。次のプログラム例を見てください。

```
      .
      .
      .
      JUMPS
RepeatLoop:
      jmp   SkipOverData
      DB    100h DUP(?)
SkipOverData:
      .
      .
      .
      dec   dx
      jnz   RepeatLoop
      .
      .
      .
```

この例では,見てわかるようにループの最後のJNZとラベル*RepeatLoop*の間が256バイト以上離れているため,JNZによって*RepeatLoop*までジャンプすることはできません。しかしJUMPS疑似命令が指定されているので,アセンブル時にエラーとはなりません。Turbo Assemblerは,実際にはこのコードを次のように認識します。

```
      .
      .
      .
RepeatLoop:
      jmp   SkipOverData
      DB    100h DUP(?)          ; CS内のデータの一時保存領域
SkipOverData:
      .
      .
      .
      dec   dx
      jz    $+5
      jmp   RepeatLoop
      .
      .
      .
```

このプログラムでは,前のプログラムのループの一番下のJNZ命令のかわりにJZとJMPのペアが使われています。

JUMPSを使えば,上記のように条件/無条件ジャンプのペアが常に生成されるわけではありません。指定した条件ジャンプによってジャンプできる場合は,条件ジャンプがそのまま使われます。次の例ではジャンプ先のラベルは1バイト分の変位でとどく距離なので,ループの一番下のJNZ命令がそのままアセンブルされます。

```
      .
      .
      .
      JUMPS
RepeatLoop:
      add   BYTE PTR [bx],1
      inc   bx
      dec   dx
      jnz   RepeatLoop
      .
      .
      .
```

前述のように，Turbo Assemblerの自動ジャンプサイズ決定によって8086の条件ジャンプの問題がすべて解決されるわけではありません。Turbo Assemblerでは，後方ジャンプ（ジャンプ命令から逆方向つまり下位アドレスへのジャンプ）に対する自動サイズ決定を，メモリや命令を少しも無駄にすることなく処理しています。

Turbo Assemblerは前方ジャンプ（ジャンプ命令から順方向つまり上位アドレスへのジャンプ）の自動サイズ決定も行ないますが，後方ジャンプのそれほど完全ではありません。良い点は，自動ジャンプサイズ決定を指定すれば，nearラベルへの前方条件ジャンプは常にアセンブルされることです。悪い点は，条件ジャンプだけでラベルに到達できても余分な**NOP**命令がいくつか挿入されることです（これはTurbo Assemblerが1パスアセンブラであることに起因します）。

前方ジャンプの自動サイズ決定が，なぜ常に最適のコードを生成しないかは少し考えれば理解できます。Turbo Assemblerが前方参照（順方向参照）をする条件ジャンプを見つけた時点では目的のラベルをまだ認識していないため，ジャンプ命令からそのラベルまでの距離を知る方法がありません。自動ジャンプサイズ決定が指定されている場合は目的アドレスが直接に読める距離にあれば条件ジャンプ（2バイト命令）を生成し，そうでない場合は無条件ジャンプと条件ジャンプ（2バイト命令+3バイト命令）のペアを生成します。Turbo Assemblerが前方条件ジャンプ命令を見つけた時点では，2バイト命令または5バイト命令のどちらが必要かまだわかりません。

しかし，以降の命令のアセンブルを続行するためにはとりあえずジャンプサイズを決定しなくてはなりません。そのためTurbo Assemblerは，安全のため，条件/無条件ジャンプのペア5バイトを確保せざるを得ないのです。そして後にジャンプ先ラベルに到達した時点で2バイト命令で十分と認識すると，条件ジャンプ命令に続いて確保しておいた5バイト領域に3つの**NOP**命令を挿入してアセンブルします。

例として次のコードをアセンブルする場合を考えます。

```
.
.
.
JUMPS
jz   DestLabel
inc  ax
.
.
.
```

JZが*DestLabel*に直接に届かない場合は，次のコードをアセンブルします。

```
.
.
.
jnz  $+5                ; 2バイト長
jmp  DestLabel          ; 3バイト長
inc  ax
.
.
.
```

JZが*DestLabel*に直接に届く場合は，次のコードをアセンブルします。

```
.
.
.
jz   DestLabel          ; 2バイト長
nop                     ; nop命令は1バイト長
nop
nop
inc  ax
.
.
.
```

ここでのキーポイントは，自動サイズ決定された前方条件ジャンプに対しては5バイトのコードを生成することと，ジャンプ先に直接到達できる自動サイズ決定された前方条件ジャンプには3個の**NOP**命令を挿入することです。これらの3個の**NOP**命令は余分なメモリと時間（8086上では各々3サイクル）を費やします。したがってプログラムのサイズと処理速度が特に重要な要因となる場合は，自動サイズ決定による前方条件ジャンプはできるだけ使わないほうがよいでしょう。

効率的なプログラムを書くためには，プログラムの重要でない部分には自動ジャンプサイズ決定を有効にし，重要な部分には自動ジャンプサイズ決定を無効にすると良いでしょう。したがって後方ジャンプには自動ジャンプサイズ決定を有効にし，前方ジャンプには自動ジャンプサイズ決定を無効にする必要があります。**JUMPS**命令の自動ジャンプサイズ決定を無効にする**NOJUMPS**命令を使えば，これが可能となります。

次のプログラムでは，自動ジャンプサイズ決定を後方ジャンプに対してのみ使い，前方ジャンプには使いません。

```
      .
      .
      .
LoopTop:
      .
      .
      .
      lodsb
      cmp  al,80h
      NOJUMPS
      jb   SaveByteValue
      neg  al
SaveByteValue
      stosb
      .
      .
      .
      dec  dx
      JUMPS
      jnz  LoopTop
      .
      .
      .
```

この例では*SaveByteValue*への前方ジャンプには2バイトの条件ジャンプを直接指定し，*LoopTop*への後方ジャンプにはTurbo Assemblerに最適のコードを選ばせています。

アセンブリの開始時では常に**NOJUMPS**が選択されています。自動ジャンプサイズ決定を使いたい場合は，**JUMPS**疑似命令でそれを明示的に有効にしなくてはなりません。

コードとデータへの前方参照

前の節では，自動ジャンプサイズ決定を使うとTurbo Assemblerが前方条件ジャンプに対していかに効率の悪いコードを生成するかを例示しました。実際に，前方参照はどんな種類のものでもTurbo Assemblerに負荷をかけます。したがって前方参照（プログラムの上位アドレスにあるラベルの参照）はできるだけ使わないことです。

その理由は, Turbo Assemblerではソースモジュールをアセンブルする際にコードを1回しかパスしないからです。ソースモジュールは先頭から最後に向かって順番に処理されます。つまりTurbo Assemblerは，はじめにモジュールの先頭の行をアセンブルし，次に2行目，3行目，……と順番にアセンブルします。考えれば明らかなように，Turbo Assemblerがアセンブルする行の順番の意味は明確ではありません。Turbo Assemblerはある行に達するまで，その行について何も知らないことになります。つまりTurbo Assemblerは前方参照では推測に頼らざるをえず，その推測は常に正しいとは限りません。推測が正しくない場合は，Turbo Assemblerは効率の悪いプログラムコードを生成します。効率のよいコードを生成できたとしても，前の行に戻ってコードを訂正しなくてはなりません。この場合は，効率の悪いプログラムコードを生成する場合よりアセンブル時間が長くなります。

次に例を示します。

```
        .
        .
        .
        jmp   DestLabel
        .
        .
        .
DestLabel:
        .
        .
        .
```

この例では, Turbo Assemblerが次の命令を認識した時点では, ラベル*DestLabel*の場所はまだわかっていません。

```
jmp   DestLabel
```

つまり*DestLabel*がnear型であるのかfar型であるのか未定です。さらにnear型だとしても，ジャンプ先に到達するために1バイト変位でよいのか2バイト変位が必要かを知る必要があります。つまりTurbo Assemblerは，アセンブルを継続するために*DestLabel*の型を推測する必要があります。

Turbo Assemblerは*DestLabel*をとりあえずfar型と推定し，far型**JMP**命令に5バイトを確保します。しかしほとんどのジャンプは3バイトのnearジャンプであるため，前方参照のnearジャンプに余分な2バイトを浪費してしまいます。もし1バイトの変位で*DestLabel*までジャンプすることができるなら，**JMP SHORT**命令で2バイトのみを確保すればそれで済みます。しかしたいていのジャンプはshortでないため，2バイトしか確保していないと，ジャンプがfarまたはnearと判明した時点でエラーとなります。

SHORT演算子または**FAR PTR**演算子でnear型を指定していない限り，Turbo Assemblerはすべての前方ジャンプに対してnear型を想定します。したがって前方ジャンプ命令には常に3バイトが確保されます。この場合は，前方ジャンプがfar型であることがわかるとエラーになります。far型ラベルへの前方ジャンプをアセンブルするには，**FAR PTR**演算子を使わなくてはなりません。**FAR PTR**を付けないと，Turbo Assemblerはデータタイプのオーバライドが必要であることをエラーとして通知します。その場合は必要な**FAR PTR**演算子を挿入すれば，再アセンブルすることができます。

一方前方ジャンプがshortであると判ると，Turbo Assemblerはショートジャンプをアセンブルし，確保されていた3バイトに**NOP**命令を1つ挿入して1バイトを無駄にします。たとえば，Turbo Assemblerは

```
      .
      .
      .
      jmp  DestLabel
DestLabel:
      .
      .
      .
```

を次のようにアセンブルします。

```
      .
      .
      .
      jmp  SHORT DestLabel
      nop
DestLabel:
      .
      .
      .
```

この例ではジャンプは正常に即座に実行されますが，命令コードのサイズが必要以上に大きくなります。**SHORT**演算子を使えば，任意の前方参照ジャンプを本当の2バイト命令に変えることもできます。しかしこの方法は，Turbo Assemblerが自動的に適切なサイズのジャンプ命令を生成する場合ほど便利ではありません。

ここで問題の原因となっているのは前方参照であることを理解してください。Turbo

Assemblerが目的のラベルまでの距離を知っていれば，最も効率よくジャンプ命令をアセンブルできるでしょう。しかし前方参照ではTurbo Assemblerはジャンプ先に到達するまでは，ジャンプ命令からジャンプ先までの距離を知ることができません。また前方参照ジャンプをアセンブルしないと，アセンブル処理をジャンプ先まで進めることができません。Turbo Assemblerは単純な推測によってアセンブル処理を続行することで，このジレンマを解決しています。しかしコードが必要以上に大きくなる可能性があります。

Turbo Assemblerはジャンプの型（**SHORT, NEAR, FAR**）がわかっている場合は，可能な限り効率的なコードを生成します。したがって，はじめからshortとわかっている前方ジャンプには**SHORT**演算子を使うのがよいでしょう（far型前方ジャンプには**FAR PTR**が必要です）。

前方参照で使うのを避けるべき命令は，ジャンプ命令だけではありません。データへの前方参照も，同じように効率の悪いコードを生成することがあります。次に例を示します。

```
        .
        .
        .
        .CODE
        .
        .
        .
        mov   bl,Value
        .
        .
        .
Value         EQU
        .
        .
        .
```

この例では，Turbo Assemblerは**MOV**命令に達した時点では，*Value*がイコール定義ラベルなのかメモリ変数なのかを知る方法はありません。*Value*がメモリ変数であれば4バイト命令が必要ですが，イコール定義（定数として使われる）ラベルであれば2バイト命令で十分です。

このような場合，Turbo Assemblerは最悪の場合を想定して**MOV**命令に4バイトを確保してアセンブルを継続します。*Value*に達した時点でメモリ変数ではなくイコール定義ラベルであることがわかると，Turbo Assemblerは**MOV**命令まで戻り，それを定数オペランド付きの2バイト命令にして，確保しておいた3番目と4番目のバイトに2個の**NOP**命令を挿入します。*Value*が**MOV**命令の前にあればTurbo Assemblerは*Value*がメモリ変数でないことを認識していたため，このようなことは起こらなかったことに注意してください。

後方参照では前方参照のような問題は起こりません。後方参照ラベルについては不明な点はありません。Turbo Assemblerは，後方参照だけを使った命令に対しては可能な限り

効率のよいコードを生成します。したがって，できるだけ前方参照は使わない方がよいでしょう。

前方参照のコールにおいても，ジャンプと同じような問題が起こると思われるかもしれません。しかし答はノーです。Turbo Assemblerは前方コールをnear型と推測するため，前方参照farコールには**FAR PTR**型オーバライドを指定する必要があります。ショートコールというものはないため，関数コールにおいては効率の悪いコードは生成されません。

多くの前方参照では効率の悪いコードは生成されず，むしろアセンブルエラーが発生します。たとえばイコール定義ラベルへの前方参照はアセンブルできません。またfar型ラベルへの前方参照は，型オーバライドを付けないとアセンブルできません。

前方参照に対して効率のよいコードを生成できる場合でも，アセンブリ速度は後方参照より遅くなります。なぜなら，Turbo Assemblerが過去に前方参照されたラベルを見つけると，そのラベルを前方参照した命令に戻ってラベルの値と型を認識したうえでアセンブルを行なうためです。このことは，そのラベルへの前方参照に型オーバライドを使った場合にも当てはまることに注意してください。

この節の結論は明白です。効率のよいコードをできるだけ速く生成させるためには，プログラムにはできる限り前方参照は使わないでください。ソースモジュール内では，データ定義はそのデータを参照するコードよりも前の位置（モジュールのはじめ）に記述してください。やむをえず前方参照を使う場合は，必らずオーバライドプレフィクスを付けてどの型のラベルを操作するのかを明確にしてください。

繰り返しブロックとマクロ命令の使用

コンピュータの得意分野の1つは繰り返し処理です。多くのDB疑似命令の値を入力したり，同じようなコードで少しずつ違うものを何度も入力することは，人間ならすぐに飽きてしまいます。しかしコンピュータは飽きるということを知りません。Turbo Assemblerの繰り返しブロックとマクロ命令を使えば，こういった単調な仕事からプログラマを解放します。

繰り返しブロック

繰り返しブロックは**REPT**疑似命令で始まり，**ENDM**ブロックで終わります。繰り返しブロック内のコードは，**REPT**疑似命令のオペランドによって指定された回数だけアセンブルされます。次に例を示します。

```
.
.
.
REPT 10
DW    0
ENDM
.
.
.
```

この例は次のような命令コードを生成します。

```
.
.
.
DW     0
DW     0
DW     0
DW     0
DW     0
DW     0
DW     0
DW     0
DW     0
DW     0
.
.
.
```

これは特に驚くことではありません。

```
DW   10 DUP (0)
```

を使っても同じことが行なえます。では繰り返しブロックと＝疑似命令を組み合わせて，0～9の整数の表を作ってみましょう。

```
          .
          .
          .
IntVal     = 0
     REPEAT 10
     DW     IntVal
IntVal     = IntVal+1
     ENDM
          .
          .
          .
```

これは次のような命令コードを生成します。

```
     .
     .
     .
DW    0
DW    1
DW    2
DW    3
DW    4
DW    5
DW    6
DW    7
DW    8
DW    9
     .
     .
     .
```

同じことを**DUP**を使ってやってみましょう。さらに，0～99の整数を得たければ**REPT**のオペランドを100に変更してください。この方法によれば，100行を入力するよりはるかに簡単です。

REPTの優れた応用例の1つに，高速の乗算，除算のために使われるテーブルの生成があります。次のプログラムは0～99の整数（BXにストアされています）と10の乗算を高速で実行して，結果をAXにストアします。

```
     .DATA
TableOfMultipleOf10   LABEL  WORD
     REPT 100
     DW   BaseVal
BaseVal    =   BaseVal+10
     ENDM
     .
     .
     .
     .CODE
     .
     .
     .
     shl  bx,1                          ; ワードサイズのテーブルを
                                        ; 参照するためにbxを2倍する
     mov  ax,[TableOfMultipleOf10+bx]   ; 10倍した結果を見る
     .
     .
     .
```

繰り返しブロック内のテキストは，**REPT**命令のオペランドで指定した回数だけ単純にアセンブルされることを覚えておいてください。繰り返しブロックを10回実行することと，繰り返しブロック内のコードをさらにあと9回コピーし，コードの10ブロックをアセンブルすることは同じです。

繰り返しブロック内には，命令を含む任意のアセンブラコードを入れることができます。次の例はDX:AX内の32ビットの符号なし数値を割算するコードを生成します。

```
.
.
.
REPEAT 4
shr    dx,1
rcr    ax,1
ENDM
.
.
.
```

繰り返しブロックはネストできます。次の例は10個の**NOP**命令を生成します。

```
.
.
.
REPT 5
REPT 2
NOP
ENDM
ENDM
.
.
.
```

繰り返しブロックと変数パラメータ

IRPと**IRPC**疑似命令によって,繰り返しブロックのブロックに対して変数パラメータを渡すことができます。

IRPは，繰り返しブロックの先頭項目内のパラメータのかわりにリストの先頭エントリを使います。同様に*n*番目の項目内のパラメータのかわりに*n*番目のエントリを使います。このような処理をリストの最後のエントリまで行ないます。次に例を示します。

```
.
.
.
IRP   PARM,<0,1,4,9,16,25>
DB    PARM
ENDM
.
.
.
```

は以下のコードを生成します。

```
.
.
.
DB    0
DB    1
DB    4
DB    9
DB    16
DB    25
.
.
.
```

IRPCも**IRP**に似ています。しかし,繰り返しブロックの各繰り返し項目に対して文字列の1文字を使うという点では異なります。次のコードは，ALが**IRPC**の2番目の引数である文字列内の文字のどれかが等しい場合にゼロフラグをセットします。

```
      .
      .
      .
      IRPC   TEST_CHAR,azklg
      cmp    al,'&TEST_CHAR&'
      jz     EndCompare
      ENDM
EndCompare:
      .
      .
      .
```

これは，以下のコードを生成します。

```
      .
      .
      .
      cmp   al,'a'
      jz    EndCompare
      cmp   al,'z'
      jz    EndCompare
      cmp   al,'k'
      jz    EndCompare
      cmp   al,'l'
      jz    EndCompare
      cmp   al,'g'
      jz    EndCompare
EndCompare:
      .
      .
      .
```

この例では，アンパサンド(&)を使って引用符内でも繰り返しブロックパラメータ*TEST_CHAR*を評価することを指定しています。アンパサンドは繰り返しブロック内で使われるマクロ命令演算子です（繰り返しブロックも，実際はマクロ命令の一種です）。**LOCAL**や**EXITM**疑似命令のような他のマクロ命令も，繰り返しブロック内で使われます。次にマクロ命令について説明しましょう。

マクロ命令

マクロ命令の基本機能は単純です。テキストのブロックつまりマクロ(macro)に名前を割り当てます。名前を割り当てておけば，後にTurbo Assemblerがソースコード内でそのマクロ名を見つけると，それに対応づけられたテキストのブロックをアセンブルしてくれます。そしてマクロ名は具体的なテキストに**展開**されます。マクロ名のマクロテキストへの置き換えは**マクロの展開**と呼ばれます。

マクロと同様に便利なのはインクルードファイルです。Turbo Assemblerは**INCLUDE**疑似命令を見つけると，その場所に指定ファイル内のテキストがあるかのように**INCLUDE**ファイルを即座にアセンブルします。同じファイルを何回でも**INCLUDE**指定することができます。その場合は，Turbo Assemblerは何回でも同じテキストをアセンブルします。

マクロ名を見つけるたびにテキストすなわちマクロボディをアセンブルするという点でマクロとインクルードは似ています。しかしマクロは必要に応じてパラメータを渡したりローカルラベルを含めることができるため，インクルードファイルよりはるかに柔軟性があります。マクロ命令のテキストはディスクから読み込む必要がないため，インクルード

ファイルよりはるかに高速です。基本的なマクロ命令の働きについてもう少し見てみましょう。

次のコードはマクロ命令*MULTIPLY_BY_4*を使った例です。このマクロはAXの値に4を掛け，結果をDX:AX にストアします。

```
        .
        .
        .
MULTIPLY_BY_4  MACRO
        sub   dx,dx
        shl   ax,1
        rcl   dx,1
        shl   ax,1
        rcl   dx,1
        ENDM
        .
        .
        .
        mov   ax,[MemVar]
        MULTIPLY_BY_4
        mov   WORD PTR [Result],ax
        mov   WORD PTR [Result+2],dx
        .
        .
        .
```

この例では，Turbo Assemblerは*MULTIPLY_BY_4*を見つけると，すぐにそのマクロボディを4個の命令に展開しアセンブルします。これは，**MOV**や**MUL**命令と同等の*MULTIPLY_BY_4*という新しい命令が定義されたことと同じです。この新しいマクロ命令は5個の8086命令から構成されますが，マクロ命令がない場合よりマクロ命令を使った方がプログラムが読みやすくなるのは確かです。

次に示すように，この例ではマクロ命令のかわりに*MultiplyBy4*という名前のサブルーチンを使うことも考えられます。

```
        .
        .
        .
MultiplyBy4    PROC
        sub   dx,dx
        shl   ax,1
        rcl   dx,1
        shl   ax,1
        rcl   dx,1
        ret
MultiplyBy4    ENDP
        .
        .
        .
```

```
     mov  ax,[MemVar]
     MULTIPLY_BY_4
     mov  WORD PTR [Result],ax
     mov  WORD PTR [Result+2],dx
     .
     .
     .
```

ではどうやってサブルーチンとマクロ命令を使いわけたらよいのでしょう？　サブルーチンを使えば一般的にコードは小さくなります。サブルーチンでは特定の作業用のコードは一度しかアセンブルされず，そのコードがプログラムの各所からコールされます。一方，マクロ命令は**CALL**命令や**RET**命令のオーバーヘッドがないため，より速いプログラムを作ることができます。また多くの類似処理のために，マクロ命令を改良して少しだけ異なるコードを生成できますが，サブルーチンではこれができません。

一般にプログラムサイズを最小限にしたいときにはサブルーチンを使い，速度と柔軟性が重要なときはマクロ命令を使えばよいでしょう。

マクロ命令はどのような柔軟性をもっているでしょうか？　マクロ命令にはパラメータを渡したり，条件アセンブル疑似命令を入れることができ，プログラマの想像力しだいでいくらでもその性能を引き出すことができます。マクロ命令に渡すパラメータは**MACRO**疑似命令のオペランドとして指定します。次の例では，*VALUE*と*LENGTH*はマクロ命令*FILL_ARRAY*に渡すパラメータです。

```
     .
     .
     .
FILL_ARRAY    MACRO    VALUE,LENGTH
     REPEAT LENGTH
     DB   VALUE
     ENDM
     ENDM
     .
     .
     .
```

マクロ命令を使うと，マクロ命令のパラメータはマクロ命令アセンブル時のオペランドとして評価されます。たとえば，*FILL_ARRAY*は以下のように評価されます。

```
     .
     .
     .
ByteArray LABEL   BYTE
     FILL_ARRAY   2,9
     .
     .
     .
```

マクロ命令の呼び出し時に使われるパラメータ（上の例では2と9）は**実パラメータ**と呼ばれます。一方，マクロ命令定義時に指定されるパラメータ（上の例では*VALUE*と*LENGTH*）は**仮パラメータ**と呼ばれます。マクロ命令が呼び出されると，マクロ命令の展開前に仮パラメータに対して対応する実パラメータの値が割り当てられます。したがって，

```
        .
        .
        .
ByteArray  LABEL   BYTE
        FILL_ARRAY    2,9
        .
        .
        .
```

は，次のようにアセンブルされます。

```
        .
        .
        .
ByteArray  LABEL     BYTE
        REPEAT 9
        DB     2
        ENDM
        .
        .
        .
```

マクロ命令定義時の仮パラメータにはアセンブル時に実パラメータが代入されるため，マクロ命令の実パラメータを変更するだけで異なるマクロ命令コードを生成できます。たとえば，*ByteArray*を長さ8バイト，初期値0FFh, *ByteArray2*を長さ100hバイト，初期値0に初期化するプログラムは次のようになります。

```
        .
        .
        .
ByteArray    LABEL   BYTE
        FILL_ARRAY  0ffh,8
ByteArray2   LABEL   BYTE
        FILL_ARRAY  0,100h
        .
        .
        .
```

仮パラメータはマクロ命令のどこにでも使えますが，仮パラメータが他のテキストと混在しているときは問題になります。次に例を示します。

```
        .
        .
        .
PUSH_WORD_REG  MACRO  RLETTER
        push  RLETTERx
        ENDM
        .
        .
        .
```

この例では，Turbo Assemblerは*RLETTERx*内にある文字列*RLETTER*が仮パラメータ名なのか，**PUSH**のオペランドの一部なのか判別できません。Turbo Assemblerはこれをオペランドの一部と認識します。しかし*RLETTERx*というメモリ変数が存在しない限り，このプッシュは正常に行なわれないでしょう。いずれにしろレジスタのプッシュはうまく達成されません。

この解決法は仮パラメータ名を一対のアンパサンド(&&)で囲むことです。Turbo Assemblerはアンパサンドで囲まれたマクロテキストを見つけると，はじめにそのテキストが仮パラメータ名かどうかをチェックします。もし仮パラメータならTurbo Assemblerはそのパラメータの値を実際の値に置き換えます（このようなテキストが仮パラメータ名でなければ，アンパサンドを無視します)。

次の例は*PUSH_WORD_REG*というマクロの展開です。

```
        .
        .
        .
PUSH_WORD_REG       MACRO   RLETTER
        push  &RLETTER&x
        ENDM
        .
        .
        .
        PUSH_WORD_REG   b
        .
        .
        .
```

この例は次のようにアセンブルされます。

```
push bx
```

アンパサンドは，仮パラメータへの参照に疑問がある場合にのみ必要です。次の例では必要ありません。

```
        .
        .
        .
PUSH_WORD_REG      MACRO     REGISTER
        push   REGISTER
        ENDM
        .
        .
        .
        PUSH_WORD_REG   bx
        .
        .
        .
```

この例も次のようにアセンブルされます。

```
push   bx
```

しかしアンパサンドを使っても問題はありません。したがってアンパサンドが必要かどうかわからないときは，いつでもこれを使うようにしてください。

マクロ命令のネスト

マクロ命令の中に繰り返しブロックを入れられることは，すでに説明してきました。マクロ命令で他のマクロ命令を呼び出すこともできます。これをマクロ命令のネストと呼びます。次に例を示します。

```
        .
        .
        .
PUSH_WORD_REG         MACRO    REGISTER
        push   REGISTER
        ENDM
        .
        .
        .
PUSH_ALL_REGS         MACRO
        IRP   REG,<AX,BX,CX,DX,SI,DI,BP,SP>
        PUSH_WORD_REG    REG
        ENDM
        ENDM
        .
        .
        .
```

この例では，マクロ命令*PUSH_ALL_REG*の中に繰り返しブロックが含まれます。繰り返しブロックは，他のマクロ命令*PUSH_WORD_REG*を呼び出しています。

マクロ命令と条件疑似命令

マクロ命令の最大の特長は，その中に条件アセンブル疑似命令を入れられることでしょう。イコール定義ラベルの状態やマクロ命令呼び出しへのパラメータによって，同じマクロ命令で異なる種類のコードをアセンブルすることができます。

先ほどのマクロ命令を使った乗算を再び例として考えてみます。この例では，新しい*MULTIPLY*マクロ命令へ渡すパラメータ*FACTOR*が2の累乗のときは，高速シフトとローテート命令を使って乗算を行ないます。それ以外のときは，**MUL**命令を使います。マクロ命令は次のようになります。

```
        .
        .
        .
MULTIPLY    MACRO     FACTOR
;
; 2の累乗(15乗～0乗)に対してFACTORをチェックする
;
IS_POWER_OF_TWO = 0
COUNT           = 15
POWER_OF_TWO    = 8000h
        REPT 16
IF POWER_OF_TWO EQ FACTOR
IS_POWER_OF_TWO = 1               ; FACTORが2の累乗の場合
        EXITM
ENDIF
COUNT           = COUNT-1
POWER_OF_TWO    = POWER_OF_TWO SHR 1
        ENDM

IF IS_POWER_OF_TWO
        sub  dx,dx
        REPT COUNT
        shl  ax,1
        rcl  dx,1
        ENDM
ELSE
        mov  dx,FACTOR
        mul  dx
ENDIF
        ENDM
        .
        .
        .
```

*MULTIPLY*は乗算が2の累乗かをチェックし，そのつど適切なコードをアセンブルします。したがってマクロ命令MULTIPLY 10は，

```
.
.
.
mov  dx,10
mul  dx
.
.
.
```

というコードにアセンブルされます。しかし，マクロ命令MULTIPLY 8は，

```
.
.
.
sub  dx,dx
shl  ax,1
rcl  dx,1
shl  ax,1
rcl  dx,1
shl  ax,1
rcl  dx,1
.
.
.
```

とアセンブルされます。

マクロ命令は実行時ではなくアセンブリ時に展開されることに注意してください。*MULTIPLY*は呼び出されるたびに新たにコードをアセンブルします。*MULTIPLY*内の**IF**疑似命令によって，どの命令がアセンブルされるかが決まります。マクロ命令とサブルーチンを混同しないでください。また，条件アセンブルを高級言語の**if**文などと混同しないでください。

EXITMで展開を停止する

前の例では疑似命令**EXITM**が使われています。**EXITM**疑似命令は，カレントのマクロ命令や繰り返しブロックの展開を停止するようTurbo Assemblerに指示します。しかし，カレントマクロ命令または繰り返しブロックが他のマクロ命令または繰り返しブロック内でネストされている場合は，ひき続きネストされているマクロ命令または繰り返しブロックが展開されます。

この例では，**EXITM**は**EXITM**疑似命令を含む繰り返しブロックの展開を停止します。しかし，繰り返しブロックを含むマクロ命令*MULTIPLY*の展開は継続します。また**REPT**と**EXITM**を使ってCにおける**while**ループに類似したものを構成しています。*MULTIPLY*の**while**ループは，どのコードをアセンブルすべきかを決めるアセンブル時のループで，実行時のループではありません。

マクロ命令内でラベルを定義する

マクロ命令内でラベルを定義するとき，マクロ命令に関する潜在的な問題が発生します。次の例ではマクロ命令*DO_DEC*の展開のたびに*SkipLabel*が定義されるため，*SkipLabel*の再定義エラーが起きます。

```
        .
        .
        .
DO_DEC     MACRO
        jcxz  SkipLabel
        dec   cx
SkipLabel:
        ENDM
        .
        .
        .
        DO_DEC
        .
        .
        .
        DO_DEC
        .
        .
        .
```

Turbo Assemblerでは，**LOCAL**疑似命令を使ってこの問題を簡単に解決できます。マクロ命令内に**LOCAL**疑似命令があると，指定された（1つまたは複数の）ラベルのスコープをそのマクロ命令に制限することができます。たとえば，**LOCAL**を使って上の例を次のようにアセンブルすることができます。

```
        .
        .
        .
DO_DEC     MACRO
        LOCAL  SkipLabel
        jcxz   SkipLabel
        dec    cx
SkipLabel:
        ENDM
        .
        .
        .
```

LOCALをマクロ命令内で使う場合は，**MACRO**疑似命令のすぐ後に置かなくてはなりません。1つの**LOCAL**疑似命令で複数のラベルを宣言できます。またマクロ命令内で複数の**LOCAL**疑似命令を使うことができます。次に例を示します。

```
      .
      .
      .
TEST_MACRO   MACRO
      LOCAL   LoopTop,LoopEnd,SkipInc
      LOCAL   NoEvent,MacroDone
      .
      .
      .
      ENDM
      .
      .
      .
```

この例で実際にローカルラベルに割り当てられる名前は，次の形式をもっています。

```
??XXXX
```

ここで*XXXX*は0～0FFFFhの16進数です。したがってプログラムでは??で始まるラベル名を使ってはなりません。Turbo Assemblerが生成するローカルラベルと重複するおそれがあります。

マクロ命令では前方参照は使えないため，マクロ呼び出しより前にマクロ定義を置かなくてはなりません。なぜならTurbo Assemblerは，前方参照のマクロ命令を見つけても何バイトを確保したらよいか判断することができません。前方参照とならない限り，マクロ命令の呼び出しはソースモジュールのどこに置いてもかまいません。

マクロ命令の中には，データ定義疑似命令，コード，セグメント疑似命令，その他任意の種類のラベル，リスティングコントロール疑似命令などの任意の有効なアセンブラ行を入れることができます。

マクロ命令用の条件アセンブル疑似命令には，**IFDIF, IFIDN, IFDIFI, IFIDNI, IFB, IFNB**があります。また，マクロ命令用の条件エラー疑似命令には，**ERRDIF, ERRIDN, ERRDIFI, ERRIDNI, ERRB, ERRNB**があります。これらの疑似命令については，このマニュアルの第5章と**リファレンスガイド**の第3章を参照してください。

マクロ命令内で使える特殊演算子は次のとおりです。

&	置換演算子
<>	リテラルテキスト文字列演算子
!	引用文字演算子
%	式評価演算子
;;	取り消しコメント

置換演算子&については，マクロ命令に関する前の節で述べました。これらの特殊演算子については，**リファレンスガイド**の第2章で詳しく説明してあります。

任意のデータ構造を扱う

Turbo Assemblerは，複雑な構造をもつデータの処理を簡単に行なうための3個の疑似命令**STRUC, RECORD, UNION**を備えています。これらの疑似命令が高級言語で使われているデータ構造疑似命令と似ていることにおそらくお気づきでしょう。実際Turbo Assemblerのデータ構造疑似命令と高級言語のデータ構造疑似命令はいくつかの類似点があります。

後で説明しますが，アセンブリ言語のデータ構造疑似命令は確かに便利ですが高級言語の疑似命令ほど洗練されていません。たとえば，アセンブリ言語は構造体の要素の名前のスコープを制限しません。したがって構造体要素のそれぞれの名前はそのソースモジュールに固有でなくてはなりません。

またCやPascalの場合とは異なり，アセンブリ言語のデータ構造疑似命令は便利ではあっても必要不可欠というものではありません。アセンブラでは，データ構造疑似命令をあえて使わなくても構造体，レコード，共用体を処理する方法が他にあります。しかしデータ構造疑似命令は便利であり，学習しておく価値はあります。

この節の説明は，Turbo AssemblerのMASMモードに適用するものです。Idealモードでは，はるかに強力なデータ構造体疑似命令の形式がサポートされます。Idealモードの詳細についは，第12章を参照してください。

Turbo Assemblerの任意のデータ構造（構造体，レコード，共用体）は，機械命令や疑似命令で前方参照されない限りソースモジュールの任意の場所で使うことができます。

STRUC疑似命令

STRUC疑似命令を使って構造体を定義することができます。これは論理的なグループに分割できるデータの処理には非常に便利です。Cに慣れている方は，**STRUC**はCの**struct**文のようなものだと考えてください。

たとえば，ある顧客の名前，年齢，収入からなる構造体を定義する場合を考えてみましょう。次にその構造体を示します。

```
CLIENT   STRUCT
NAME     DB     'Name goes here ....'
AGE      DW     ?
INCOME   DD     ?
CLIENT   ENDS
```

*CLIENT*構造体は3個のフィールドを持ちます。*NAME*フィールドには最大20文字の名前

が，*AGE*フィールドには16ビット値の年齢が，*INCOME*フィールドには32ビット値の収入がそれぞれ格納されます。

*CLIENT*構造体は次のように使うことができます。

```
      .
      .
      .
CLIENT   STRUC
NAME     DB     'Name goes here....'
AGE      DW     ?
INCOME   DD     ?
CLIENT   ENDS
      .
      .
      .
      .DATA
MisterBark     CLIENT   <'John Q. Bark',32,10000>
      .
      .
      .
      .CODE
      .
      .
      .
      mov  ax,[MisterBark.Age]
      mov  bx,OFFSET MisterBark
      mov  ax,WORD PTR [bx.INCOME]
      mov  dx,WORD PTR [bx.INCOME+2]
      .
      .
      .
```

この例では説明すべきことがたくさんあります。第1に，構造体定義は**ENDS**疑似命令で終わることに注意してください。この疑似命令はセグメント定義を終えるときに使ったものと同じです。セグメント定義の中に構造体定義をネスト（入れ子）させることは可能です。次の例はデータセグメント内に構造体を定義しています。

```
      .
      .
_Data    SEGMENT  WORD PUBLIC 'DATA'
      .
      .
Test      STRUC
      .
      .
Test      ENDS
      .
      .
_Data     ENDS
      .
      .
```

第2に，*CLIENT*という名の新しいデータ型があるかのように構造体*CLIENT*の変数*MisterBark*を定義していることに注意してください。*CLIENT*構造体にSIZE演算子を指定すれば，構造体のサイズとして26という値が得られます。

*MisterBark*の定義では，不等号カッコ内に3個のパラメータを指定しています。これらのパラメータは, *MisterBark*の対応するフィールドの初期値になります。文字列'John Q. Bark'は*NAME*フィールドの初期値, 32が*AGE*フィールドの初期値, 10,000が*INCOME*フィールドの初期値です。

構造体変数の定義時に，そのすべてのフィールドに初期値を指定する必要はありません。たとえば，次の定義は*MisterBark*のどのフィールドも初期化しません。

```
MisterBark   CLIENT  <>
```

次の定義は*INCOME*フィールドのみを初期化します。

```
MisterBark   CLIENT  <,,19757>
```

初期化するフィールドがなくとも不等号カッコは必要です。

メモリ変数の定義時に初期値を指定しなくても，各フィールドの初期値をセットする方法が2通りあります。構造体の定義時に特定のフィールドに値を指定すると，それはそのフィールドに割り当てられるデフォルト値になります。たとえば構造体の定義時に特定のフィールドに疑問符（?）を指定すると，そのフィールドのデフォルト値は0になります。

たとえば次のプログラムでは, *MisterBark*の定義時に*MisterBark*の1つのフィールド(*NAME*フィールド)にのみ初期値を指定しています。*CLIENT*構造体の定義時には，*AGE*フィールド用の初期値が指定されています。したがって，それが*MisterBark*の*AGE*フィールドに割り当てられるデフォルト値となります。*INCOME*フィールドのどちらの場所にも値は指定されていないので，*INCOME*フィールドは0に初期化されます。

```
      .
      .
      .
CLIENT   STRUC
NAME     DB    'Name goes here....'
AGE      DW    21
INCOME   DD    ?
CLIENT   ENDS
      .
      .
      .
      .DATA
MisterBark     CLIENT   <'John Q. Bark'>
      .
      .
      .
```

この例では，*NAME*フィールドは'John Q. Bark'に初期化され，*AGE*フィールドは21に初期化され，*INCOME*フィールドは0に初期化されます。また*MisterBark*の定義時に指定された*NAME*フィールドの初期値は，*CLIENT*構造体の定義時に指定されたデフォルト値にオーバライドすることに注意してください。

構造体の配列は，次のように**DUP**で初期化することができます。

（リスト1行 R10_69.lst）

この例では，*CLIENT*という型の52個の構造体よりなる配列*Clients*を定義し，各構造体をデフォルト値に初期化します。

以前の構造体の例に戻ってください。ピリオド（.）演算子という新しい演算子があるのにお気づきでしょう。ピリオド演算子は，メモリアドレシングのプラス演算子の別の形式です。以下の命令はすべて同じ機能です。

```
.
.
.
mov  ax,[bx.AGE]
mov  ax,[bx].AGE
mov  ax,[bx+AGE]
mov  ax,[bx]+AGE
.
.
.
```

ピリオド演算子の役割は2つあります。1つはピリオド演算子を使うC言語の表記と合わせるため，もう1つは構造体フィールドをアクセスしていることを明確にするためです。ピリオドまたはプラスのどちらでも好きな方を使ってください。

STRUC疑似命令で定義された構造体フィールドは，実際は構造体フィールドのオフセットに対するイコール定義ラベルです。*CLIENT*と*MisterBark*を定義した先の例では，次の2行は同じです。

```
.
.
.
mov  [MisterBark.AGE],ax
mov  [MisterBark+20],ax
.
.
.
```

さらに，次のようなイコール定義を使うこともできます。

```
      .
      .
      .
AGE_FIELD  EQU  20
      .
      .
      .
      mov  [MisterBark+AGE_FIELD],ax
      .
      .
      .
```

STRUCを使うことのメリットとデメリット

STRUCを使う理由の1つに，フィールドがデータ型を決定することにあります。最初の例の*AGE*は構造体要素なのでTurbo Assemblerは*MisterBark.AGE*がワードサイズであることを認識しますが，2番目の例の*MisterBark*＋*AGE*は固有のサイズを持ちません。

2番目の理由として，定数オフセットやイコール定義のセットを変更するより構造体定義を変えるほうがはるかに簡単ということがあります。たとえば*NAME*フィールドを30文字の長さに変更したい場合にやらなけれならないことは，*CLIENT*内の*NAME*エントリを変更することだけです。イコール定義を使っていたら，*AGE*と*INCOME*フィールド両方のオフセットを手で計算し，変更しなくてはなりません。大きな構造体の場合は作業量はもっと多くなるでしょう。

3番目の理由は，STRUCを使えばデータ構造（構造体）の定義と初期化を簡単に行なうことができるからです。

要するに，STRUCはデータ構造の定義やアクセスを簡単に行なうための保守しやすい手段です。しかしアセンブラのデータ構造では，Cのデータ構造ほどエラー対策が施されていません。たとえば，レジスタを使ってアセンブルデータ構造を指すと，Turbo Assemblerはそのレジスタの内容がその型のデータ構造を指すポインタなのか認識することができません。次のコードは，BXに0をロードします。しかしTurbo Assemblerは有効な*CLIENT*データ構造がオフセット0にあるか否かを知る方法はありません。

```
.
.
.
mov  bx,0
mov  dx,[bx.AGE]
.
.
.
```

これはアセンブリ言語の問題ではなく，アセンブリ言語の性質を反映するものです。プログラミングの種類として，自由なプログラミングと安全なプログラミングの2種類があるとすれば，アセンブラプログラミングの種類は自由なプログラミングといえます。Turbo Assemblerは，構造体に対しては限られた範囲のエラーチェックしか行なわないことに注意してください。プログラマは正しいポインタがロードされていることを，自分の責任において確かめなければなりません。

固有の構造体フィールド名

構造体フィールド名は事実上ラベルであるため，ソースモジュール内で固有でなくてはなりません。たとえばソースモジュール内で*CLIENT*を定義した場合は，そのモジュール内の任意の場所のみならず，他の構造体の中でさえ*INCOME*という名のラベルを使うことはできません。*INCOME*は22という値をもつラベルにすぎず，もちろん1つのソースモジュール内で複数の同名ラベルを使うことはできません。次の例は*AGE*を再定義しようとしているため，エラーになります。

```
       .
       .
       .
CLIENT     STRUC
NAME       DB     'Name goes here ....'
AGE        DW     ?
INCOME     DD     ?
CLIENT     ENDS
       .
       .
       .
AGE   EQU  21
       .
       .
       .
```

構造体のネスト

構造体はネストできます。次に例を示します。

```
       .
       .
       .
      .DATA
       .
       .
```

```
AGE_STRUC STRUC
YEARS      DW  ?
MONTHS     DW  ?
AGE_STRUC ENDS
     .
     .
CLIENT     STRUC
NAME       DB  'Name goes here ....'
AGE        AGE_STRUC <>
INCOME     DW        ?
CLIENT     ENDS
     .
     .
MisterBark    CLIENT   <>
     .
     .
     .CODE
     .
     .
     mov  dx,[MisterBark.AGE.MONTHS]
     mov  si,OFFSET MisterBark
     mov  cx,[si.AGE.YEARS]
     .
     .
```

この例では，*CLIENT*構造体の中で*AGE*という名の*AGE_STRUC*構造体をネストして，*CLIENT*構造体の*MisterBark*内の*AGE*フィールドの*MONTHS*と*YEARS*を参照しています。

構造体の初期化

構造体の初期化に関して注意すべきことがいくつかあります。第1は，構造体の文字列フィールドをそのフィールドより長い文字列で初期化しようとすると，アセンブルエラーとなります。

第2に，文字列によって初期化できるフィールドの種類は文字列フィールドだけです。次の例はたとえ*TEXT*がスペースに初期化されていても，正しくアセンブルできません。

```
     .
     .
TEST STRUC
TEXT DB   30 DUP (' ')
TEST ENDS
     .
     .
TStruc    TEST <'Test string'>
     .
     .
```

この理由はTurbo Assemblerは*TEXT*を30文字の文字列としてではなく，30個のスペースとして認識しているからです。次の例はアセンブルできます。

```
      .
      .
      .
TEST STRUC
TEXT DB   'String goes here ............'
TEST ENDS
      .
      .
      .
TStruct   TEST <'Test string'>
      .
      .
      .
```

第3に，1つの構造体フィールドに属する複数の要素を定義することはできます。しかしその構造体の定義時には1つのフィールドに対しては1つの要素しか初期化できません。次の例は*TestStruc*を定義し，フィールド*A*の1バイト目は1に，フィールド*B*の1バイト目は2にそれぞれ初期化します。しかし各フィールドの2バイト目は20h（スペース）に初期化します。

```
      .
      .
      .
T     STRUC
A     DB    0ffh,0ffh
B     DB    0ffh,0ffh
T     ENDS
      .
      .
      .
TestStruct T  <1,2>
      .
      .
      .
```

この節では，STRUC疑似命令のMASMモードについて説明しました。Idealモードでは**STRUC**疑似命令はかなり強力で，高級言語の構造体に匹敵する性能を備えています。Idealモードについては第12章を参照してください。

RECORD疑似命令

RECORD疑似命令を使って，1バイトまたは1ワード内のビットフィールドを定義するこ

とができます。ビットフィールド定義を使えば，任意の数のビットフィールドをとり出すためのマスクや任意のビットフィールドを右寄せするためのシフトカウントが可能となります。**RECORD**疑似命令はPascalの**record**文とは何の関係もありません。

たとえば1ビットフラグ3つと12ビットの値を含むデータ構造を定義する場合は，次のように**RECORD**疑似命令を使うことができます。

```
TEST_REC   RECORD   FLAG1:1,FLAG2:1,FLAG3:1,TVAL:12
```

この例では，*FLAG1, FLAG2, FLAG3*という名の3個のフラグと*TVAL*という名の1つのデータフィールドを定義します。各フィールドのコロンの後の値は，フィールドサイズをビット数で表わします。各フラグのサイズは1ビット，*TVAL*のサイズは12ビットです。

レコード内におけるフィールドの配置は少し複雑です。最初のフィールド*FLAG1*はレコードの左端（最上位）ビットに，2番目のフィールド*FLAG2*はレコードの左から2番目のビットに，*TVAL*はレコードの残りの右側12ビットに割り当てられます。ただしレコードのサイズは15ビットのため，ワード中の1ビットは使用されないことになります(レコードは常に8ビットまたは16ビット長です)。一般規則として，レコードの内容はバイトまたはワード内で常に右詰めにされて配置されます。

上の説明は少し複雑です。わかりやすくするために次に例を挙げて説明します。*TEST_REC*型のレコードを，次のような行で定義します。

```
TRec   TEST_REC   <1,0,0,52h>
```

この例は*TEST_REC*というレコード型をもつ*TRec*という変数を定義します。不等号カッコ内の値は，対応フィールドの初期値です。*TRec*の*FLAG1*フィールドは1に，*FLAG2*と*FLAG3*フィールドは0に，*TVAL*フィールドは52hにそれぞれ初期化されます。図10.1に*TRec*の4つのフィールドの配置と初期値を示します。

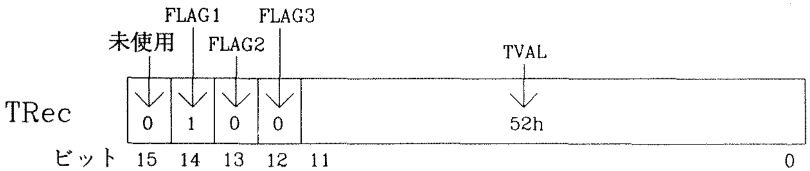


図10.1　TRec のフィールドの配置と初期値

レコードの全長（全フィールドの総計）が8ビット以下の場合，レコードは1バイト内にストアされます。それ以外の場合はレコードは1ワード内にストアされます。16ビットより長いレコードは，80386用アセンブルが有効にされている場合を除いてサポートされません。80386用アセンブルが有効になっている場合は，最大32ビットのサイズのレコードが使

えます。

レコード変数の初期化は，構造体変数の初期化とよく似ています。レコード変数の定義時にレコードフィールドの初期値を指定した場合は，前の例に示したとおりフィールドはその値に初期化されます。

レコード変数の定義時にレコードフィールドの初期値を指定しない場合は，デフォルト値には2通りの値をとります。レコード型の作成時は，必要に応じて特定のまたはすべてのフィールドにデフォルト値を指定することができます。次に例を示します。

```
TEST_REC   RECORD   FLAG1:1=1,FLAG2:1=0,FLAG3:1,TVAL:12=0FFFH
```

この例では，*FLAG1*に1, *FLAG2*に0,*TVAL*に0FFFhのデフォルト値をそれぞれ指定します。*FLAG3*には明示的なデフォルト値を指定していません。明示的なデフォルト値のない場合は0になり，*FLAG3*のデフォルト値は0になります。

次の例は*TEST_REC*というレコードと*TRec*という変数を定義します。

```
        .
        .
        .
        .DATA
        .
        .
        .
TEST_REC   RECORD   FLAG1:1=1,FLAG2:1=0,FLAG3:1,TVAL:12=0fffh
        .
        .
        .
TRec TEST_REC    <,1,,2>
        .
        .
        .
```

この例では各フィールドは次のように初期化されます。

■*FLAG1*は1に初期化される。
■*FLAG2*は1に初期化される。
■*FLAG3*は0に初期化される。
■*TVAL*は2に初期化される。

レコード変数*TRec*の全体の値は6002hです。レコード変数の定義時に指定された初期値は，レコード型の定義時に指定された初期値をオーバライドすることに注意してください。

レコード型は他のデータ型と非常によく似ています。たとえば，レコード型に**SIZE**演算子を使うことができますし，レコード配列を**DUP**演算子を使って定義することもできま

す。次の例は*TEST_REC*というレコード型の90個のレコードを持つ配列を宣言します。

```
TRecArray TEST_REC   90 DUP(<1,1,1,0>)
```

STRUCフィールド名と同じように，レコードフィールド名はラベルです。ラベルはソースモジュール内で1回しか定義できないため，すべてのレコードフィールド名はソースモジュール内で固有でなくてはなりません。

レコードのアクセス

レコードの定義方法やレコード内のフィールド配置のようすがわかりました。次にレコードへのアクセス方法を説明します。レコードのフィールドにアクセスするには，次のように構造体のフィールドにアクセスするのと同じ方法が使えるように思われます。

```
mov  al,[TRec.FLAG2]    ; これはできない!!!
```

しかし，実際はこのようにアクセスすることはできません。8086は8ビットまたは16ビットの幅のメモリオペランドでのみ動作するため，たとえば1ビットフィールドをレジスタにロードすることはできません。レコードフィールドに関して行なえる操作は，バイト単位でそのサイズを決めること，右詰めにするのに何ビットシフトする必要があるかを決めること，ビットを取り出すためにマスクをすることだけです。つまり，8086では直接レコードフィールドを操作することはできませんが，Turbo Assemblerは**AND**や**SHR**などの命令でレコードフィールドの操作をサポートしています。

レコードフィールドの値は，何ビット分シフトしたらそのフィールドを右詰めにする（つまりフィールドのビット0をレコードのビット0に配置する）ことができるかを表わす値です。次に例を示します。

```
.
.
mov  al,FLAG1
mov  ah,TVAL
.
.
```

この例では，ALに14，AHに0がそれぞれロードされます。

```
.
.
mov  ax,[TRec]
mov  cl,FLAG1
shr  ax,cl
.
.
```

この例では，AX内の*TRec*の*FLAG1*フィールドが右詰めされます。

レコード型自身の値はバイト値またはワード値です。この値は，初期値を持つレコードを定義することによって生成されます。次に例を示します。

```
mov   ax,TEST_REC <1,1,1,0fffh>
```

この例ではAXに7FFFhをロードします。この値は，*TEST_REC*型レコードを初期値<1, 1, 1, 0FFFh>で定義した場合に得られる値です。上の例のようにレコード型*TEST_REC*をAXにロードすることと，次の例のようにレコード変数*TRec*をAXにロードすることの違いに注意してください。

```
      .
      .
TEST_REC   RECORD   FLAG1:1=1,FLAG2:1=0,FLAG3:1,TVAL:12=0fffh
      .
      .
TRec TEST_REC  <,1,,2>
      .
      .
      .CODE
      .
      .
      mov   ax,[TRec]
      .
      .
```

この例では，変数*TRec*の値6002hがAXにロードされます。

WIDTH演算子

WIDTH演算子は，レコード，またはレコードフィールドのサイズをビット数で返します。次の例はALに*TEST_REC*レコードサイズ（ビット数15）をストアします。

```
mov   al,WIDTH TEST_REC    ; TEST_RECレコードのサイズ(ビット数)
```

次の例は，各フラグフィールドの幅1をAL, AH, BLに，*TVAL*フィールドの幅12をBHにストアします。

```
.
.
mov   al,WIDTH FLAG1
mov   ah,WIDTH FLAG2
mov   bl,WIDTH FLAG3
mov   bh,WIDTH TVAL
.
.
```

MASK演算子

MASK演算子は，**AND**命令でレコードまたはレコードフィールドを取り出すためのマスクパターンを返します。次に例を示します。

```
mov  ax,MASK TEST_REC
```

この例はAXに7FFFhをストアします。次に別の例を示します。

```
.
.
.
mov  ax,MASK TEST_REC
mov  dx,[Trec]
and  dx,ax
.
.
.
```

この例は，レコード*TRec*の値のうち，*TEST_REC*の構成ビットでないビット15をマスクしてDXにストアします。

MASK演算子は，レコードフィールドを個別に取り出すために使うとさらに便利です。次の例は，*TRec*の*FLAG3*フィールドがセットされているか否かを調べます。

```
.
.
mov  ax,[Trec]
and  ax,MASK FLAG3
jz   Flag3NotSet
.
.
```

AND命令のかわりに**TEST**命令を使うと，レジスタを破壊しないで済むことに注意してください。次の例は，レジスタやメモリの内容を変えないで前例と同じテストをします。

```
.
.
test [TRec],MASK FLAG3
jz   Flag3NotSet
.
.
```

シフト命令に関連して**MASK**演算子を使えば，レコードフィールドを操作するのに便利です（後述します)。

レコードの用途

レコードが何であるか，またレコードがどのように使われるかを説明しましたが，どのような場合に実際にレコードを使ったら良いのでしょうか？　レコードはそれほど頻繁に使われるわけではありません。1つのバイトまたはワード内に複数のデータフィールドがコード化されているときにレコードを使うと便利です。BIOSで使われる変数の中にはレコード構造を持つものもあります。たとえば，次のようにIBM PCのBIOSの装置フラグ変数の下位バイトは，装置に関する情報（どのビデオアダプタがアクティブか，使用されているフロッピィディスクドライブの数など）を保持します。

```
EQ_FLAG RECORD NUMDISKS:2,VIDEO:2,RSRVD:2,MATHCHIP:1,AREDISKS:1
```

- *NUMDISKS*は，インストールされているフロッピィディスクドライブの数マイナス1です。
- *VIDEO*は，現在アクティブなディスプレイアダプタの種類です。
- *RSRVD*は，他のIBMマイクロコンピュータで他の目的に使うために確保されたフィールドです。
- *MATHCHIP*は，8087などの数値演算コプロセッサがインストールされていれば1です。
- *AREDISKS*は，フロッピィディスクドライブがインストールされていれば1です。

次の関数は，*EQ_FLAG*レコードやレコード演算子を使って，BIOSの装置フラグ変数のディスプレイアダプタフィールドの設定値を返します。

```
;
; BIOSの装置フラグ変数のディスプレイアダプタの設定を返すファンクション
;
; 入力:なし
;
; 出力
;    AL = 0 (ディスプレイアダプタが選択されていない)
;           1 (40×25カラーディスプレイアダプタが選択されている)
;           2 (80×25カラーディスプレイアダプタが選択されている)
;           3 (80×25モノクロディスプレイアダプタが選択されている)
;
; 破壊されるレジスタ: AX,CL,ES
;
EQ_FLAG RECORD NUMDISKS:2,VIDEO:2,RERVD:2,MATHCHIP:1,AREDISKS:1
;
```

```
GetBIOSEquipmentFlag    PROC
      mov   ax,40h
      mov   es,ax            ; ESにBIOSデータセグメントへのポインタをロード
      mov   al,es:[10h]      ; 装置フラグの下位ビットをロード
      and   al,MASK VIDEO    ; ディスプレイアダプタフィールドのみ取り出す
      mov   cl,VIDEO         ; ディスプレイアダプタフィールドを右詰めにする
                             ; ために右シフトすべきビット数
      shr   al,cl            ; ディスプレイアダプタを右詰め
      ret
GetBIOSEquipmentDFlag   ENDP
```

次の関数は，BIOSの装置フラグのディスプレイアダプタフィールドに指定された値をセットする補足的な関数です。

```
      .
      .
      .
      .DATA
Counter   FLEX_COUNT   <?,?>
      .
      .
      .
      .CODE
      .
      .
      .
      mov   [Counter.Count16],0ffffh
LoopTop:
      .
      .
      .
      dec   [Counter.COUNT16]
      jnz   LoopTop
      .
      .
      .
      mov   [Counter.COUNT8],255
ShortLoopTop:
      .
      .
      .
      dec   [Counter.COUNT8]
      jnz   ShortLoopTop
      .
      .
      .
```

この節では，MASMモードの**RECORD**疑似命令について説明しました。**RECORD**疑似命令は，IdealモードとMASMモードではその機能が若干異なります。Idealモードについては第12章を参照してください。

UNION疑似命令

UNION疑似命令は，1つのメモリアドレスを複数のデータ型として参照することを可能にします。**UNION**はCの**union**文と似ています。

たとえば，あるカウンタを8ビットカウンタとして使ったり16ビットカウンタとして使ったりする場合は，次のようにカウンタを2種類の共用体として宣言することができます。

```
        .
        .
FLEX_COUNT     UNION
COUNT8          DB     ?
COUNT16         DW     ?
FLEX_COUNT     ENDS
        .
        .
```

STRUCと**UNION**の定義は，**ENDS**で終わることに注意してください。

この例のように*FLEX_COUNT*共用体が定義されていると，*FLEX_COUNT*を2つの用途のカウンタとして使用することができます。

```
        .
        .
        .DATA
Counter    FLEX_COUNT    <?,?>
        .
        .
        .CODE
        .
        .
        mov   [Counter.Count16],0ffffh
LoopTop:
        .
        .
        dec   [Counter.COUNT16]
        jnz   LoopTop
        .
        .
        mov   [Counter.COUNT8],255
ShortLoopTop:
        .
        .
        dec   [Counter.COUNT8]
        jnz   ShortLoopTop
        .
        .
```

STRUCと同様，共用体の参照にもピリオド演算子が使われています。プラス演算子も使

うことができます。共用体フィールドを使って変数を参照することは，型オーバライドを使うことと同じです。したがって，この例は次のように書き直すことができます。

```
        .
        .
        .
        .DATA
Counter    DW    ?
        .
        .
        .
        .CODE
        .
        .
        .
        mov   WORD PTR [Counter],0ffffh
LoopTop:
        .
        .
        .
        dec   WORD PTR [Counter]
        jnz   LoopTop
        .
        .
        .
        mov   BYTE PTR [Counter],255
ShortLoopTop:
        .
        .
        .
        dec   BYTE PTR [Counter]
        jnz   ShortLoopTop
        .
        .
        .
```

共用体が型オーバライドよりも優れている点は，型オーバライドを憶えるより正しい共用体要素名を使う方がはるかに簡単なことにあります。また，共用体変数の複数モード処理は変数の定義を見れば明確なため，共用体を使ったプログラムはわかりやすく保守しやすいのです。

共用体の中で共用体と構造体の両方をネストすることができます。次の例では，共用体を使うことによって,4バイトのメモリ変数をダブルワードサイズのセグメント：オフセットポインタとして，またはワードサイズのオフセット変数とワードサイズのセグメント変数のペアとしてアクセスすることができます。

```
        .
        .
        .
SEG_OFF     STRUC
POOFF        DW    ?
PSEG         DW    ?
SEG_OFF     ENDS
        .
        .
        .
PUNION      UNION
DPTR         DD      ?
XPTR         SEG_OFF    <>
PUNION      ENDS
        .
        .
        .
        .CODE
        .
        .
        .
        mov   [bx.XPTR.POFF],si
        mov   [bx.XPTR.PSEG],ds
        .
        .
        .
        les   di,[bx.DPTR]
        .
        .
        .
```

STRUCやRECORDの場合と同じように，UNIONで定義されたフィールド名は通常のラベルとして扱われるためスコープの制限はありません。したがって共用体フィールド名はソースモジュール内で固有でなくてはなりません。

この節では，MASMモードにおけるUNION疑似命令について説明してきました。IdealモードのUNION疑似命令はかなり強力で，高級言語の共用体に匹敵する機能を備えています。Idealモードについては第12章を参照してください。

セグメント疑似命令

第4章で，簡略セグメント疑似命令の使用法について説明しました。そして，正常動作するプログラムを作るために，標準セグメント疑似命令についても詳しく学んできました。次に標準セグメント疑似命令について1つ1つ詳しく説明すると同時に，簡略セグメント疑似命令の役割についても解説します。コードセグメントやデータセグメントを使ったサンプルプログラムも示します。マルチセグメントプログラム（複数セグメントを持つプログラム）がどのように動作するのかが理解できます。

簡略セグメント疑似命令は標準セグメント疑似命令よりも使いやすい反面，機能的には劣ることを念頭に置いてください。以下の節で説明する標準セグメント疑似命令は，**SEGMENT, GROUP, ASSUME**です。

SEGMENT疑似命令

SEGMENT疑似命令はセグメントを開始するために使います。各**SEGMENT**疑似命令の後には，セグメントを終了させるために対応する**ENDS**命令がなくてはなりません。簡略セグメント疑似命令と異なり，**SEGMENT**は各セグメントの属性を指定することができます。

SEGMENT疑似命令の完全な形式は，次のようになっています。

```
name  SEGMENT  align  combine use 'class'
```

ここで，*align, combine, use, class*はすべてオプション（任意指定）項目です。次に各フィールドについて説明します。

セグメント名とアラインメントフィールド

*name*はセグメントの名前です。セグメント名はラベルなので，ソースモジュール内で固有でなくてはなりません。セグメントを終了するときは，**ENDS**疑似命令でそのセグメント名を指定しなくてはなりません。

*align*はセグメントを開始するメモリ境界を指定します。以下に有効なアラインメントを示します。

■**BYTE**は，次のバイトアドレスを指定します。

■**DWORD**は，次のダブルワードアドレスを指定します。

■**PAGE**は，次のページアドレス（256バイト単位）を指定します。

■**PARA**は，次のパラグラフアドレス（16バイト単位）を指定します。

■**WORD**は，次のワードアドレスを指定します。

アラインメントを指定しないと，パラグラフアラインメントが使われます。

バイトアラインメントを使うと，もっともコンパクトなプログラムを作ることができます。16ビットプロセッサはワード単位のデータを効率よく処理できるため，ワードアラインメントは16ビットコンピュータに適しています。同じように，ダブルワードアラインメントは32ビットコンピュータに適しています。64Kバイトのフルサイズのセグメントには，パラグラフアラインメントが必要です。

combineフィールド

*combine*は，プログラムの構成モジュールのリンクのときに，別モジュールの同名セグメントどうしの結合方法（結合型）を指定します。*combine*には以下の型を指定できます。

AT	**PRIVATE**
COMMON	**PUBLIC**
MEMORY	**STACK**

少し後の「簡略セグメント疑似命令」の節で高級言語で使う結合型について説明してあります。

AT結合型は，メモリの特定アドレスからセグメントを開始します。実際のコードは生成しません。**AT**セグメントは，ROM BIOSデータセグメントやディスプレイメモリなどのメモリにアクセスするためのテンプレート（ひな型）として使われます。次に例を示します。

```
      .
      .
      .
VGA_GRAPHICS_MEMORY SEGMENT AT 0A000h
BitMapStart   LABEL   BYTE
VGA_GRAPHICS_MEMORY  ENDS
      .
      .
      .
```

```
mov  ax,VGA_GRAPHICS_MEMORY
mov  es,ax
ASSUME  ES:VGA_GRAPHICS_MEMORY
mov  di,OFFSET BitMapStart
mov  cx,08000h
sub  ax,ax
cld
rep  stosw
.
.
.
```

この例はIBM PCのVGAグラフィックススクリーンをクリアします。

COMMON結合型は，あるセグメントの開始点と他のすべての同名セグメントがアラインされるべき開始点を指定します。これによって，セグメントは互にオーバレイとなります。セグメントの合計サイズは，これらのセグメントのうちの最大サイズとなります。**COMMON**結合型を使うためには，そのセグメントを参照する各モジュール内に**COMMON**セグメントを定義するファイルをインクルードしておきます。そうすれば，全モジュールが効率よく同一のセグメントを共有することができます。

PUBLIC結合型は，同名セグメントどうしを連結するようリンカに指示します。セグメントは効率よくまとめられ，より大きいセグメントになります。セグメントの合計のサイズは，これらのすべてのセグメントのサイズの和となります。すべてのセグメントと同じように，**PUBLIC**セグメントの合計のサイズも64Kバイトを超えることはできません。**PUBLIC**は複数のモジュールが同じセグメントを共有し，かつ各モジュールが固有の変数を定義する場合に使われます。**PUBLIC**セグメント内の変数は，**GLOBAL**疑似命令によってモジュール間で共有されることがよくあります。

MEMORY結合型は**PUBLIC**と同じです。

STACK結合型は，同名の全セグメントを連結してEXEファイルを作成し，プログラム実行時にはSS:SPがこのセグメントの終わりを指すようにリンカに指示します。これはスタックセグメントに対してのみ使われる特殊な結合型です。

PRIVATE結合型は，セグメントを他のどのセグメントにも結合しないようリンカに指示します。この結合型では，特定のモジュールに対してローカルなセグメントを定義することができ，同名セグメントを他のモジュール内で使っても重複セグメントについて心配する必要がなくなります。結合型を指定しない場合のセグメントのデフォルトの結合型は**PRIVATE**です。

useフィールドとclassフィールド

SEGMENT疑似命令の*use*フィールドは80386でのみ使います。*use*フィールドについては第11章を参照してください。

*class*フィールドは，リンカがセグメントを配置する順序を指定します。特定のクラスの全セグメントは，ソースコード内での順序にかかわらず，メモリの連続ブロック内に置かれます。「簡略セグメント疑似命令」の節に高級言語で使うクラスを示します。簡単のためにこれらの規定に従うことができます（次の節で，セグメントオーダつまりセグメントの順序付けについて説明します)。

セグメントのサイズ，型，名前，ネスティング

クラス内のセグメントの合計サイズは，実行時に利用できるメモリサイズによってのみ制限されます。しかし，各セグメントのサイズは64Kバイトを超えることはできません。

クラス型は引用符で囲まなければならないことに注意してください。またクラス型はソースモジュール内で固有でなくてはなりません。つまり，1つのモジュール内で使うラベルとクラス型とが同じ名前になってはなりません。

1つのソースモジュール内で同一セグメントを複数回定義できます。この場合，どの定義も同一セグメントを参照しているとみなされます。ただし，これらの定義はすべて同じ属性を指定する必要があります。そうしないと，Turbo Assemblerではエラーとなります。

このエラーを防ぐ簡単な方法は，ソースモジュール内でのセグメントの最初の定義時にのみ属性を指定することです。属性のない再定義セグメントに対しては，そのセグメントの初回の定義時に指定された属性が自動的に使われます。

セグメントはネストできます。これは前のセグメント終了前に，別のセグメントを定義できることを意味します。次に例を示します。

```
       .
       .
DataSeg    SEGMENT PARA PUBLIC 'DATA'
       .
       .
DataSeg2   SEGMENT PARA PRIVATE 'FAR_DATA'
       .
       .
DataSeg2   ENDS
       .
       .
DataSeg    ENDS
       .
       .
```

ネスティングは一般に便利とは限りませんが，少なくともマクロ命令内で使うと便利です。マクロ命令内でセグメントを定義するには，いったんカレントセグメントを終了させてから再開始しなければなりません。このためにはカレントセグメント名がわかっていなければなりません。しかし，カレントセグメント名はマクロ命令の環境下では必らずしも明白ではありません。これに対して，セグメントのネストを使うと，カレントセグメント名を知らなくてもセグメントを定義することができます。次に例を示します。

```
      .
      .
TEST MACRO
      .
      .
TesTSeg   SEGMENT WORD PRIVATE 'FAR_DATA'
      .
      .
TesTSeg   ENDS
      .
      .
      ENDM
      .
      .
```

ネストしたセグメントが終わると，Turbo Assemblerのアセンブル処理はネスト開始時にアクティブだったセグメントへ戻ります。

セグメントの順序づけ

.EXEファイル内でのセグメントの順序については，気にかける必要はありません。.EXEファイル内のセグメントの順序はたいてい問題になりませんし，セグメント順が問題になる場合は，たいてい高級言語コンパイラまたは**DOSSEG**疑似命令が簡単に解決するからです。高級言語とリンクする場合は，たいていその言語のコンパイラがセグメント順をコントロールするでしょう。純粋なアセンブラプログラムを書いているのであれば，セグメントは以下のようなマイクロソフト社の標準セグメント順となります。

■**CODE**クラスのセグメント

■**DGROUP**の構成セグメントでない**CODE**以外のクラスのセグメント

■**DGROUP**の構成セグメント（次の順序）

- **STACK**と**BSS**以外のクラスのセグメント
- **BSS**クラスのセグメント
- **STACK**クラスのセグメント

リンカによるセグメントの順序を見たければ，/sコマンドラインを使ってください。TLINKは詳細なセグメントマップファイルを作るので，マップファイルを見ることができます。

高級言語をリンクするのでもなく**DOSSEG**疑似命令を使うのでもない場合は，セグメントはどのように順序付けられるのでしょう？　通常，プログラマはその答えを知る必要はありません。しかしセグメントの順序が問題になる場合の答えを下に記述します（これは多少複雑です）。

DOSSEGなどを使って明示的なセグメントの順序付けを行なわない場合は，リンカはクラスのすべてのセグメントをグループ化します。この際，セグメントのクラスは**SEGMENT**疑似命令のクラスフィールドで指定されます。セグメントのグループは，リンカが見つけた順序に単純に.EXEファイル内に配置されます。つまり，.OBJファイルをロードする際に，リンカが最初に見つけたセグメントクラスを.EXEファイルの先頭に，2番目のクラスを2番目に，という具合に順番に配置します。つまり，.OBJファイルがリンクされる順序は，.EXEファイル内のセグメントの順序と一致します。

これで，セグメントはクラスによって大枠で順序付けられたことになります。では，各クラス内ではセグメントはどのように順序付けされるでしょうか？　これらも，リンカが見つけた順番に.EXEファイル内に置かれます。ここでのポイントは，.OBJファイルのリンクされる順序および各.OBJファイル内でのセグメントの順序です。Turbo Assemblerは，.OBJファイル内でのセグメントの順序付けに関して，2通りの方法をとります。

.SEQ疑似命令は，ソースファイル内の順序と同じ順序でセグメントを.OBJファイル内に置くようTurbo Assemblerに指示します（シーケンシャルセグメントオーダーといいます)。この順序づけの方法では，ソースモジュール内のセグメンドの順序が.EXEファイル内のセグメントの順序に影響します。シーケンシャルセグメントオーダはTurbo Assemblerのデフォルトモードです。**.ALPHA**疑似命令を指定していない限り，**.SEQ**疑似命令を省略した場合もシーケンシャルセグメントオーダが使われます。

.ALPHA疑似命令は，セグメントを.OBJファイル内にアルファベット順に置くようTurbo Assemblerに指示します(アルファベットセグメントオーダといいます)。この順序づけの方法では，ソースモジュール内のセグメントの順序は.EXEファイル内のセグメントの順序と無関係です。これは従来のアセンブラで使われているデフォルトモードです。したがって，アセンブラプログラムを正常に動かすために**.ALPHA**を使わなくてはならない場合もあります。

上記の説明をまとめると次のようになります。セグメントの大枠の順序付けはクラスによって行ない，クラス内ではセグメントの出現順に順序づけを行ないます。クラス内のセグメントの順序は，.OBJファイルのリンクの順番および**.SEQ**疑似命令と**.ALPHA**疑似命令によって指定できます。**.SEQ**を指定すると，ソースモジュール内のセグメントの順序は.EXEファイル内のセグメントの順序と一致します。

このようにセグメント順序づけは単純な問題ではありませんが，プログラマはセグメントの順序を心配する必要はないでしょう。いずれにせよ，通常は高級言語コンパイラや**DOSSEG**疑似命令がセグメント順序づけを自動的に行ないます。

GROUP疑似命令

GROUP疑似命令は複数のセグメントを結合して，1つの論理グループにまとめるために使われます。この際，すべてのセグメントは1つのセグメントレジスタに基づいてアドレシングされます。

2個のセグメント内のデータをアクセスするプログラムを例にとります。通常は他のセグメントにアクセスするたびに，セグメントレジスタを設定し，新たに**ASSUME**を設定しなければならないでしょう。これは時間を浪費して煩わしいことです。セグメントを結合して*DataGroup*という名の1つのグループにまとめ，*DataGroup*をDSにセットしてDSを*DataGroup*に仮定し，任意のセグメントにアクセスする方が簡単です。次に例を示します。

```
      .
      .
      .
DataGroup  GROUP   DataSeg1,DataSeg2
      .
      .
      .
DataSeg1   SEGMENT  PARA PUBLIC 'DATA'
MemVar1    DW    0
DataSeg1   ENDS
      .
      .
      .
DataSeg2   SEGMENT  PARA PUBLIC 'DATA'
MemVar2    DW    0
DataSeg2   ENDS
      .
      .
      .
      mov   ax,DataGroup
      mov   ds,ax
      ASSUME     DS:DataGroup
      .
      .
      .
      mov   ax,[MemVar1]
      mov   [MemVar2],ax
      .
      .
      .
```

1つのセグメント名とPUBLIC結合型を使えばもっと簡単に同じことができるのに，なぜグループを使うのでしょう？　純粋なアセンブラプログラムでは必要に応じてグループを使うことはできますが，その必要はあまりありません。グループは主としてアセンブラコードを高級言語とリンクする際に使われます。特に高級言語で**DGROUP**を使うと，スタック，初期化されたnearデータ，初期化されないnearデータ，定数セグメントなどを特定のセグメントレジスタに基づいてアクセスできるようになります。

グループに関する主要な規則の1つとして，すべてのセグメントは1つのセグメントレジスタに基づいてアクセスしなければならないため，グループ内の全セグメントは1つの64Kバイトセグメントの中に納まっていなくてはなりません。前の節で述べたように，セグメント順序づけは多くの要素に依存しているため，気をつけないとセグメントが離れてしまうこともあります。もっとも安全な方法は，グループ内の全セグメントは同じクラスとして宣言し，モジュールの先頭でそれらをまとめて定義することです。

高級言語とリンクする場合またはプログラム内の任意の場所ですでに**DOSSEG**疑似命令を使った場合は，**DGROUP**内のセグメントがまとめられていることを確認する必要はありません。このような場合は，リンカは全セグメントを**DGROUP**内で連続配置します。

同一グループ内の全セグメントは64Kバイトセグメント内に納まっていなくてはなりませんが，リンクの際に連続する必要はありません。グループの構成セグメントの間にグループ化されていないセグメントを挿入することも可能です。

注意：グループを使う場合は，セグメントをロードしてグループ名をポイントする際は常に**ASSUME**でグループ名を想定する必要があります。そうしないと，セグメントレジスタがたとえグループの先頭を指していても，Turbo Assemblerはグループの先頭からのオフセットではなくセグメントの先頭からのオフセットを生成します。次の例は，**DGROUP**が先に定義されているのでエラーとなります。

```
.
.
.
mov   ax,DGROUP
mov   ds,ax
ASSUME  DS:Stack      ; 不正オフセットを生成する
.
.
.
```

正しくは，次のようにしてください。

```
.
.
.
mov  ax,DGROUP
mov  ds,ax
ASSUME    DS:DGROUP
.
.
.
```

つまり，あるグループを指すようにセグメントレジスタをセットする場合は，そのグループの構成セグメントではなく必ずそのグループを指すようにしてください。

MASM（マイクロソフト社のマクロアセンブラ）では，グループに**OFFSET**演算子を使うとバグが発生します。このバグは，グループ内のラベルアドレスにデータを初期化するときにも発生します。Turbo Assemblerでも互換性を考慮して，このバグを再生します。このバグに対処するには，ラベルを**OFFSET**演算子とともに使う場合またはラベルを使ってデータを初期化する場合は，ラベルにグループオーバライドプレフィクスを必ず付けることです（詳細については第5章の「オペランドおよびデータテーブルにグループオーバライドを付け忘れる」の節を参照してください）。

ASSUME疑似命令

ASSUME疑似命令は，セグメントレジスタがどのセグメントまたはグループを指しているかをTurbo Assemblerに通知します。これは，セグメントレジスタを実際にセットすることではないことに注意してください。セグメントレジスタを実際に設定するには，**MOV**命令を使わなくてはなりません。**ASSUME**の目的は，メモリ参照の有効性をチェックし，メモリアクセスの際に必要に応じてセグメントオーバライドプレフィクスを自動的に付加するようにすることです。

CSを指定した**ASSUME**疑似命令は，ソースモジュールの先頭に入れておかなければなりません。そうすれば，Turbo Assemblerはジャンプ，コールの対象となる命令やプログラムの開始アドレスがどのセグメント内にあるのか知ることができます。

さまざまなセグメントレジスタを指定した**ASSUME**疑似命令を，ソースモジュール内に必要なだけ挿入することができます。セグメントレジスタに設定されたセグメントは，必要に応じていつでも変更できます。1つの**ASSUME**疑似命令で，任意のまたはすべてのセグメント設定を変更できます。

セグメント名，グループ名，または**SEG**演算子でラベルから取り出したセグメントのどれかで，セグメントレジスタの設定を指示することができます。さらに**NOTHING**キーワ

ードを使って，任意のまたはすべてのセグメントレジスタがどのセグメントも指さないようにTurbo Assemblerに指示することができます。

次に**ASSUME**の使用例を示します。

```
Stack     SEGMENT   PARA STACK 'STACK'
     DB    512 DUP (0)
Stack     ENDS
TGROUP    GROUP     DataSeg1,DataSeg2
DataSeg1 SEGMENT   PARA PUBLIC 'DATA'
      .
      .
      .
DataSeg1     ENDS
DataSeg2     SEGMENT   PARA PUBLIC 'DATA'
      .
      .
      .
DataSeg2     ENDS
      .
      .
      .
DataSeg3     SEGMENT   PARA PUBLIC 'DATA'
MemVar       DW    0
      .
      .
      .
DataSeg3     ENDS
      .
      .
      .
CodeSeg      SEGMENT   PARA PUBLIC 'CODE'
     ASSUME    CS:CodeSeg,DS:TGROUP,SS:Stack,ES:NOTHING
ProgramStart:
     mov     ax,TGROUP
     mov     ds,ax
     ASSUME    DS:TGROUP
     .
     .
     .
     mov     ax,SEG MemVar            ; DataSeg3と同じ
     mov     es,ax
     ASSUME    ES:SEG MemVar
     .
     .
     .
     push    ds
     pop     es
     mov     ax,CodeSeg
     mov     ds,ax
     ASSUME    DS:CodeSeg,ES:TGROUP
     .
     .
     .
CodeSeg      ENDS
     END     ProgramStart
```

ASSUME疑似命令がグループを参照する場合は，指定セグメントレジスタはそのグループの先頭を指していると仮定されます。しかし**ASSUME**疑似命令がグループの構成セグメントを参照する場合は，セグメントレジスタは（そのグループではなく）セグメントの先頭を指していると仮定されます。セグメントレジスタは一般にグループの先頭を指すようセットされ，グループの構成セグメントを指すようにはセットされないので，これはトラブルの原因となる場合があります。次の例では，DSは**TGROUP**の先頭を指しているにもかかわらず，**ASSUME**文でDSが*DataSeg2*の先頭を指すよう誤って指示しているため，間違ったメモリアドレスの内容をAXにロードしてしまいます。

```
      .
      .
      .
TGROUP    GROUP     DataSeg1,DataSeg2
DataSeg1  SEGMENT   PARA PUBLIC 'DATA'
      .
      .
      .
DataSeg1  ENDS
DataSeg2  SEGMENT   PARA PUBLIC 'DATA'
MemVar    DW    0
DataSeg2  ENDS
      .
      .
      .
CodeSeg   SEGMENT   PARA PUBLIC 'CODE'
      ASSUME    CS:CodeSeg
      .
      .
      .
      mov   ax,TGROUP
      mov   ds,ax
      ASSUME    DS:DataSeg2           ; 誤り!!!（TGROUPが正しい）
      mov   ax,[MemVAR]               ; TGROUPではなくDataSeg2のオフセット
                                      ; からデータをロードする
      .
      .
      .
```

簡略セグメント疑似命令を使う場合は，通常Turbo Assemblerが自動的に適当なセグメントを想定するため，**ASSUME**を使用する必要はありません。しかし簡略セグメント疑似命令を使ってセグメントレジスタのどれかを変更する場合は，適切な**ASSUME**疑似命令を実行しなければなりません。次の例は，DSレジスタが**.DATA**セグメント，**.CODE**セグメント，**.FARDATA**セグメント，**.DATA**セグメントの順序にポイントするように設定します。

```
.
.
.
.DATA
.
.
.
.FARDATA
.
.
.
.CODE
mov  ax,@data
mov  ds,ax
ASSUME   DS:@data
.
.
.
mov  ax,@code
mov  ds,ax
ASSUME   DS:@code
.
.
.
mov  ax,@fardata
mov  ds,ax
ASSUME   DS:@fardata
.
.
.
mov  ax,@data
mov  ds,ax
ASSUME   DS:@data
.
.
.
```

前述のように，（直前に発行した**ASSUME**疑似命令に基づいて動作している）Turbo Assemblerがメモリ変数にアクセスする必要がある場合は，メモリアクセスの際に常にセグメントオーバライドプレフィクスを付けるよう**ASSUME**疑似命令で指示することができます。下の例では，Turbo Assemblerは*MemVar*にアクセスする命令に自動的にES:オーバライドを付けます。なぜなら**ASSUME**疑似命令によって，*MemVar*の存在するセグメントをDSが正しく指定できないようにされているためです。

```
     .
     .
     .
DataSeg   SEGMENT   PARA PUBLIC 'DATA'
MemVar    DB    ?
     .
     .
     .
DataSeg   ENDS
     .
     .
     .
CodeSeg   SEGMENT   PARA PUBLIC 'CODE'
     ASSUME  CS:CodeSeg,DS:NOTHING,ES:DataSeg
     .
     .
     .
     mov  ax,DataSeg
     mov  ds,ax
     mov  es,ax
     mov  [MemVar],1
     .
     .
     .
```

したがって，プログラマは**ASSUME**疑似命令が常にセグメントレジスタの実際の設定に対応していることを確かめる必要があります。

簡略セグメント疑似命令

簡略セグメント疑似命令については，第4章である程度説明しました。しかし，さまざまな簡略セグメント疑似命令がどのようなセグメントを生成するのかはまだ説明していません。プログラマは通常このことは知る必要はありませんが，簡略セグメント疑似命令と標準セグメント疑似命令とを混同しないためにも，一応はこれについて知っておく必要があります。

.CODE, .DATA, .DATA?, .STACK, .CONST, .FARDATA, .FARDATA?の各疑似命令により作成されるセグメントとセグメントグループは，**.MODEL**疑似命令で選択されるメモリモデルによって異なります（メモリモデルは第4章で説明しました）。メモリモデルと簡略セグメント疑似命令によって生成されるセグメントの関係を以下の表に示します。

表10.1　タイニィメモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	_TEXT	WORD	PUBLIC	'CODE'	DGROUP
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

表10.2　スモールメモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

表10.3　ミディアムメモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	name_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

表10.4 コンパクトメモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	FAR_DATA	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

表10.5 ラージまたはヒュージメモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

これまでの章でとりあげた簡略セグメント疑似命令を使ったプログラムでは，**ASSUME, GROUP, ENDS**疑似命令を必要としないことにお気づきでしょう。**.MODEL**疑似命令は選択メモリモードに対して自動的に適切な**ASSUME**疑似命令を実行し，その際上の表で示すセグメントを想定します。**.MODEL**疑似命令はまた，上の表に示す**DGROUP**のグループ定義を実行します。

ENDSはセグメントを終了します。簡略セグメント疑似命令（**.CODE**や，**.DATA**など）によって新たなセグメントが開始すると，（もしあれば）カレントセグメントは自動的に終了します。

次にもう少し複雑な簡略セグメント疑似命令の**.DATA?, .CONST, .FARDATA, .FARDATA?**について見てみましょう。**.FARDATA**は，疑似命令のうちで純粋アセンブラプログラムで使われる唯一のものです。他の疑似命令は，高級言語のセグメントの使用法に合わせるためのものです。

.DATA?は初期化されていないnearデータを入れるためのDGROUP内のセグメントを開始します。.DATAセグメントと.DATA?セグメントは同じグループ内にあるため，.DATA?を使用する理由はありません。また疑問符を使って.DATAセグメント内に初期化されていないデータを定義する理由もありません。ただし，高級言語の規定にしたがっている場合はこの限りではありません。

.CONSTはnear型定数を入れるためのDGROUP内セグメントを開始するもので，.DATA?と同じカテゴリーに入ります。高級言語の規定にしたがっている場合を除いて，定数データは.DATAに入れて.CONSTを使用しないようにすることもできます。

.FARDATAは，特定のソースモジュールに固有のfarデータセグメント（他のモジュールと共有されないセグメント）を作成するのに使います。そのセグメントの名前はFAR_DATAですが，結合型がPRIVATEのため，他のセグメントとは結合されません。.FARDATAを使うとモジュール内に最大64Kバイトのローカルデータメモリを定義できます。.FARDATA疑似命令を使う場合は，以下のようにセグメントレジスタがfarデータセグメントをポイントするよう設定することはいうまでもありません。

```
      .MODEL   SMALL
      .DATA
InitValue DW   0
      .FARDATA
MemArray  DW   100 DUP (?)
      .CODE
      .
      .
      .
      mov  ax,@data
      mov  ds,ax
      mov  ax,@fardata
      mov  es,ax
      mov  ax,[InitValue]
      ASSUME   ES:@fardata
      mov  di,OFFSET MemArray
      mov  cx,100
      cld
      rep  stosw
      .
      .
      .
```

定義済みラベル@fardataには.FARDATA疑似命令で定義されたセグメント名が含まれることに注意してください。

.FARDATAで定義されたセグメントは，（.DATAで定義されたセグメントとは異なり）他のモジュールと共有されることはありませんが，GLOBALを使えば.FARDATAセグメント内の特定の変数を他のモジュールと共有することができます。次の例では変数*MemVar*は他のモジュールから参照可能となっています。

```
        .MODEL      SMALL
        .FARDATA
        GLOBAL      MemVar:WORD
MemVar      DW    0
        .
        .
        .
```

他のモジュールは，次のように変数*MemVar*を参照することができます。

```
.MODEL      SMALL
GLOBAL      MemVar:WORD
.DATA
.
.
.
.CODE
.
.
.
mov   ax,SEG MemVar
mov   ds,ax
ASSUME      DS:SEG MemVar
mov   ax,[MemVar]
.
.
.
```

*MemVar*のGLOBAL宣言はどのセグメント宣言よりも前にあることに注意してください。変数のグローバル宣言は変数のセグメント内またはすべてのセグメントの外で行なわなければならないため，これは必要なことです。どのモジュールも定義によって他のモジュールの**.FARDATA**セグメントを共有できないため，*MemVar*の宣言は全セグメントの外で行なわなくてはなりません。

.FARDATA?は**.FARDATA**に非常によく似ていますが，**FAR_BSS**という名のプライベートセグメントを作成する点で異なっています。**FAR_BSS**セグメントは高級言語が初期化しないfarデータ用に使います。高級言語とリンクするのでなければ，**.FARDATA**で定義されたセグメント内で初期化しないfarデータを定義できるため，**.FARDATA?**は使う必要はありません。**.FARDATA**セグメントによって，64Kバイトのfarメモリを新たに確保することもできます。特定のモジュールに固有な64Kバイトを超えるfarメモリが必要な場合は，標準セグメント疑似命令を使う方がよいでしょう。

.FARDATA?を使うと，定義済みラベル**@fardata?**は**.FARDATA**で定義されたセグメント名を表わすことになります。これは**ASSUME**疑似命令での使用やセグメントレジスタの設定に適しています。

マルチセグメントプログラムのサンプル

次のサンプルプログラムは，2個のコードセグメントと2個のデータセグメントを持っています。これは決してマルチセグメント（複数セグメント）プログラムの最適な例ではありません。しかしこのマニュアルでは，何百，何千ステップに及ぶプログラムを載せる余地はありません。このプログラムを見れば，データセグメントの切り替え，セグメント：オフセットポインタの設定，他セグメント内のプロシージャまたは関数の呼び出しなどについての感触をつかむことができるでしょう。

以下にサンプルプログラムを示します。

```
;
; 複数コードセグメントとデータセグメントの使い方を示すプログラム
;
; コンソールから文字列を読み込み，一方のデータセグメントへストアし，
; 他のデータセグメントへコピーする．次に文字列を小文字に変換し，
; コンソールに表示する．
; 文字列の読み込み，表示，コピーには他のコードセグメント内の関数を使う．
;
Stack   SEGMENT   PARA STACK 'STACK'
    DB  512 DUP (?)
Stack   ENDS

MAX_STRING_LENGTH   EQU  1000

SourceDataSeG  SEGMENT   PARA PRIVATE 'DATA'
InputBuffer    DB   MAX_STRING_LENGTH DUP (?)
SourceDataSeg  ENDS

DestDataSeg    SEGMENT   PARA PRIVATE 'DATA'
OutputBuffer   DB   MAX_STRING_LENGTH DUP (?)
DestDataSeg    ENDS

SubCode  SEGMENT   PARA PRIVATE 'CODE'
    ASSUME    CS:SubCode
;
; コンソールから文字列を読み込むサブルーチン
; 文字列は改行コードで終わる．改行コードはCR/LFのペアに変換し改行する．
; 文字列の終端には0を付加する．
;
; 入力:
;     ES:DI - 文字列のストア先アドレス
;
; 出力:なし
;
; 破壊されるレジスタ: AX,DI
;
```

```
GetString PROC FAR
GetStringLoop:
    mov  ah,1
    int  21h                  ; 次の文字をロード
    stosb                     ; ストア
    cmp  al,13                ; 改行コードか?
    jnz  GetStringLoop        ; No....処理続行
    mov  BYTE PTR es:[di],10
    mov  BYTE PTR es:[di+1],0 ; 文字列の終わりにLFと0を付ける
    ret
GetString ENDP
;
; 文字列を小文字に変換してコピーするサブルーチン
;
; 入力:
;   DS:SI - コピーする文字列のアドレス
;   ES:DI - コピー先アドレス
;
; 出力:なし
;
; 破壊されるレジスタ: AL, SI, DI
;
CopyLowercase   PROC FAR
CopyLoop:
    lodsb
    cmp   al,'A'
    jb    NotUpper
    cmp   al,'Z'
    ja    NotUpper
    add   al,20h              ; 大文字は小文字に変換
NotUpper:
    stosb
    and   al,al               ; 文字列の終わりは0か?
    jnz   CopyLoop            ; NO...次の文字をコピー
    ret
CopyLowercase  ENDP
;
; コンソールに文字列を表示するサブルーチン
;
; 入力:
;   DS:SI - 表示文字列へのポインタ
;
; 出力: なし
;
; 破壊されるレジスタ: AH,DL,SI
;
DisplayString  PROC FAR
DisplayStringLoop:
    mov  dl,[si]              ; 次の文字をロード
    and  dl,dl                ; 文字列の終わりの0か?
    jz   DisplayStringDone    ; Yes...終わり
    inc  si                   ; 次の文字をポイント
    mov  ah,2
    int  21h                  ; 文字を表示
    jmp  DisplayStringLoop
```

```
DisplayStringDone:
    ret
DisplayString   ENDP
SubCode   ENDS

Code SEGMENT   PARA PRIVATE 'CODE'
    ASSUME    CS:Code,DS:NOTHING,ES:NOTHING,SS:Stack
ProgramStart:
    cld                         ; ストリング命令のポインタレジスタの
                                ; カウントアップを設定
;
; コンソールから文字列を読み込みInputBufferへストアする.
;
    mov  ax,SourceDataSeg
    mov  es,ax
    ASSUME   ES:SourceDataSeg
    mov  di,OFFSET InputBuffer
    call GetString              ; コンソールから文字列を読み込み
                                ; ES:DIの指すアドレスへストア
;
; LFコードを出力して改行する.
;
    mov  ah,2
    mov  dl,10
    int  21h
;
; InputBufferからOutputBufferへ文字列をコピーする.
; このとき大文字を小文字に変換する.
;
    push es
    pop  ds
    ASSUME    DS:SourceDataSeg
    mov  ax,DestDataSeg
    mov  es,ax
    ASSUME    ES:DestDataSeg
    mov  si,OFFSET InputBuffer       ; DS:SIからロード
    mov  di,OFFSET OutputBuffer      ; ES:DIへストア
    call CopyLowercase               ; 小文字に変換
;
; 小文字の文字列を表示する
;
    push es
    pop  ds
    ASSUME    DS:DestDataSeg
    mov  si,OFFSET OutputBuffer
    call DisplayString              ; DS:SIの文字列をコンソールに表示
;
; 終了
;
    mov  ah,4ch
    int  21h
Code ENDS
    END  ProgramStart
```

この例では，サブルーチンがメインプログラムの前にあることに注意してください。サブルーチンとメインプログラムは別のコードセグメント内にあるため，このように配置することによって前方参照を避けています。Turbo Assemblerは前方参照のfarジャンプをアセンブルできないので，メインプログラムを先にすると各サブルーチンコールに**FAR PTR**オーバライドを付けなくてはなりません。プログラムがこの例のように構成されていればサブルーチンはすべて後方参照になり，Turbo Assemblerは自動的にサブルーチンのfarコールを生成することができます。

その他については，プログラムはきわめて単純でわかりやすくなります。サブルーチンはデータへのセグメント：オフセットのポインタを使っており，メインプログラムは必要に応じてDSとESに別のデータセグメントをセットしています。文字列を小文字に変換してコピーする際の，ストリング命令の使い方に注意してください。**LODS**はデフォルトとしてDSを使い，**STOS**はデフォルトとしてESを使います。これらの命令は，2つのセグメントに同時にアクセスしなくてはならない場合に使うのが最適です。

第11章

80386と他のプロセッサ

第10章までは，8086プロセッサ用のアセンブリ言語プログラミングに焦点を当てて解説してきました（8088は基本的には8086に8ビット外部データバスが付いたものなので，8086について解説すれば8088にも言及したことになります。なお，8088はIBM PC/XTに使われています）。Turbo Assemblerは8086の他にも，iAPx86として知られる8086のスーパセットプロセッサファミリおよび8087のスーパセット数値演算コプロセッサのファミリもサポートします。

iAPx86ファミリのうち最も興味深いのは，何といっても80386です。このプロセッサはパーソナルコンピュータにミニコン並みの能力を与えるものです。iAPx86ファミリの各製品は基本的には8086を拡張・改良したものです。ここでは，iAPx86ファミリのさまざまなプロセッサ用のプログラミングについて説明します。

最初に，80186プロセッサと80286プロセッサが8086の機能をどのように拡張しているのかについて説明し，次に80386のプログラミングを説明します。ここでは，Turbo Assemblerが80386の機能をどのように使っているのかということと，80386のプログラミングで使われる新しいセグメント型について説明します。また80386の新しいレジスタ，アドレシングモード，命令を紹介します。その後，16ビット命令，32ビット命令，セグメントを同時に扱うTurbo Assemblerの強力な機能にも言及し，80386のサンプルプログラムもいくつか紹介します。最後に80287と80387数値演算コプロセッサが8087の機能をどのように拡張しているかについても簡単に説明します。

アセンブラコード内でプロセッサの型を切り換える

Turbo Assemblerは，特に指定のない限り8086コードのみをサポートします。Turbo Assemblerが他のiAPx86ファミリもサポートするためには，適切な疑似命令を使う必要があります。以下のプロセッサ型指定の疑似命令を使って，プログラムのアセンブル時にサポートするプロセッサの型をTurbo Assemblerに指示します。

.186	**.286C**	**.287**	**.386C**	**.387**	**.8087**
.286	**.286P**	**.386**	**.386P**	**.8086**	

これらの疑似命令は，アセンブラソースファイルの任意の場所に挿入でき，ただちに効力を持ちます。ソースファイル内には，複数のプロセッサ型指定の疑似命令を入れることができます。ソースファイルの任意の場所では，直前に指定されたプロセッサが現在有効なカレントプロセッサとなります。

.8086疑似命令を使えば，いつでも8086のアセンブルのみサポートするようにTurbo Assemblerに指示することができます（この章の以降では，8086についての説明はすべて8088にも適用されます）。次の例では，はじめに8086の命令を，次に80386の命令を，最後に再び8086の命令を使って2つの32ビット値を加えています。

```
        DOSSEG
        .MODEL      SMALL
        .CODE
Add32         PROC
        mov   ax,[bp+4]          ; ソース1の下位ワードをロード
        mov   dx,[bp+6]          ; ソース1の上位ワードをロード
        mov   bx,[bp+8]          ; ソース2の下位ワードをロード
        mov   cx,[bp+10]         ; ソース2の上位ワードをロード
        .386                     ; 80386レジスタを使って加算する
        shl   eax,16
        mov   ax,dx
        rol   eax,16             ; 32ビットのソース1をEAXにロード
        mov   dx,cx
        shl   edx,16             ; 32ビットのソース2をEDXにロード
        mov   dx,bx
        add   eax,edx            ; ソース1とソース2を加える
        rol   eax,16
        mov   dx,ax              ; 結果の上位ワードをDXへストア
        shr   eax,16             ; 結果の下位ワードをAXへストア
        .8086
        ret
Add32           ENDP
        END
```

80186と80188

80186は，ほとんど8086と同様なiAPx86ファミリのプロセッサです。80186は8086のすべての命令の他，いくつかの新しい命令と8086命令の拡張版をサポートしています。さらに80186は多く処理，特にメモリアドレス計算を8086よりかなり高速で実行するため，8086用のプログラムを8086より相当速く実行することができます。

80188は80186とプログラムの互換性があります。この両プロセッサの違いは外部データバスの幅のみです。80186の外部データバスは16ビットで，80188では8ビットです。

80186用のアセンブルを有効にする

Turbo Assemblerで80186のアセンブルを有効にするには，**.186**疑似命令を使います。つぎに80186の新命令と拡張命令を見てみましょう。80186の命令についての詳細は，リファレンスガイドの第3章を参照してください。

ところで，8086はこれから説明する命令を認識できないことに注意してください。つまり，80186の新命令または拡張命令の1つでも使っているプログラムを8086で実行することはできません。

新しい命令

80186の新しい命令は次のとおりです。

BOUND	**INS**	**OUTS**	**PUSHA**
ENTER	**LEAVE**	**POPA**	

PUSHAとPOPA

PUSHAと**POPA**は，8個の汎用レジスタをプッシュおよびポップするための効果的な命令です。**PUSHA**は，8個の汎用レジスタをAX, CX, DX, BX, SP, BP, SI, DIの順番でスタックにプッシュします。**POPA**は，**PUSHA**の逆の操作を行ないます（DI, SI, BP, BX, DX, CX, AXの順番でスタックからポップします)。ただし**POPA**ではSPはポップされま

せん。そのかわり，SPには16（PUSHAによってスタックにプッシュされたレジスタのトータルバイト長）が加えられ，PUSHAでプッシュされたSPの値はPOPAによりスタックから捨てられます。セグメントレジスタ，フラグレジスタ，命令ポインタは，PUSHAやPOPAの影響を受けません。

次に例を示します。

```
      .186
      .
      .
      .
SampleFunction PROC
      pusha
      .
      .
      .
      popa
      ret
SampleFunction ENDP
      .
      .
      .
```

この例では，たったの2つの命令で8個の汎用レジスタすべてをスタックに退避します。それぞれのレジスタを別々にプッシュ，ポップすると16命令が必要になります（PUSHAやPOPAなどの80186に固有な命令を使う場合は，必らず.186疑似命令を指定して80186用のアセンブルを有効にしてください）。

PUSHAはPUSHを8回別々に実行するよりは高速ですが，PUSHを3～4回実行するよりは遅いことに注意してください。すべてのレジスタを退避するのでなければ，PUSHを使って必要なだけレジスタを退避するのがよいでしょう。POPAとPOPの関係も同様です。

ENTERとLEAVE

ENTERとLEAVEはスタックフレームの確保と開放のために使われます。スタックフレーム内では，引き渡されたパラメータとローカル（自動）変数はBPレジスタに基づいてアクセスできます。ENTERとLEAVEは，アセンブラ関数とCなどのスタック指向言語をリンクする際に特に便利です（アセンブラ関数とTurbo C, Turbo Pascal, Turbo Basic, Turbo Prologのインターフェースについては，それぞれ6, 7, 8, 9章を参照してください）。

ENTER命令は，呼び出し側ルーチンのBPを退避し，新しいスタックフレーム内に渡された引数があればBPにその開始アドレスをセットします。さらに，必要に応じてローカル変数用のスペースを確保するためにSPを調節したり，高級言語のスタックフレームへのポインタブロックを新しいスタックフレーム内へコピーしたりします。

LEAVE命令は，**ENTER**で実行した処理をすべて元に戻します。つまりBPとSPの両レジスタを**ENTER**実行前の状態に戻します。

次の関数例では，**ENTER**によってCと互換性のあるスタックフレーム(ローカル変数用の20バイト)を確保し，**LEAVE**によってスタックフレームを開放して呼び出し側プログラムのスタックフレームを復帰しています。

```
        .
        .
        .
SampleFunction PROC
        enter 20,0
        .
        .
        .
        leave
        ret
SampleFunction ENDP
        .
        .
        .
```

ENTERの第1オペランドは，新しいスタックフレーム内のローカル変数用スペースのバイト数を指定する16ビット即値で，第2オペランドは，関数(これに対して新しいスタックフレームを作成する)のネスティングレベルを指定する8ビット即値です。呼び出し側プログラムのスタックフレームから新しいスタックフレーム内にコピーされるスタックフレームポインタの数は，基本的にはこのオペランドで指定されます。

呼び出し側プログラムに戻るには，**LEAVE**の後に**RET**命令が必要なことに注意してください。**LEAVE**はカレントスタックフレームを開放するだけで，リターンはしません。

ENTERと**LEAVE**は呼び出し側プログラムのどのレジスタも退避しません。退避するには，**PUSH**と**POP**または**PUSHA**と**POPA**のペアを使ってください。

BOUND

BOUND命令は，16ビット値がメモリの連続する2ワードで指定される符号付き数値範囲内にあることをチェックします（範囲の上限値は，下限値のすぐ上位のアドレスにストアされます)。上限値と下限値はともに符号付き数値として扱われ，最大−32768～＋32767の範囲内の値を指定することができます。上限値と下限値に一致する値も，指定範囲内の値とみなされます。

通常，**BOUND**は特定の配列範囲の外へのアクセスを防止するために使われます。次の例は，BXの値が0～99の範囲内にあるかどうかをチェックした後，BXを100バイトの配列*TestArray*へのインデックスとして使います。

```
    .
    .
    .
    .DATA
TestArrayBounds  LABEL  DWORD
    DW    0                    ; 配列の下限値(範囲内に含む)
    DW    99                   ; 配列の上限値(範囲内に含む)
TestArray  DB   100 DUP (?)
    .
    .
    .
    .CODE
    .
    .
    .
    mov   ax,@data
    mov   ds,ax
    .
    .
    .
    bound bx,[TestArrayBounds]
    mov   al,[TestArray+bx]
    .
    .
    .
```

BXが範囲内にない場合は，INT 5が生成されます。もちろんBOUNDを使うためには，INT 5の割り込みハンドラが用意されていなければなりません。

BOUNDの第1オペランドは，範囲チェックの対象となる値をストアした16ビット汎用レジスタです。第2オペランドは，範囲を表わすダブルワードです。このダブルワードの下位ワードは符号付き16ビット下限値，上位ワードは符号付き16ビット上限値です。

BOUNDのわかりにくい点は，値が範囲外でINT 5が生成されたときにプッシュされる命令ポインタは次の命令を指すのではなく，INT 5を生成したBOUND命令を指すことです。INT 5ハンドラがIRET命令を実行する前に範囲外条件を訂正しないと，同じBOUND命令が別のINT 5を生成してしまい，無限ループに陥ります。つまり，BOUND命令が生成したINT 5ハンドラは，メッセージを出力してIRET命令を実行せずにプログラムを終了するか，IRET命令実行前に範囲外条件を訂正しなければなりません。

INSとOUTS

INSとOUTS命令は，I/Oポートとメモリの間の効率的なデータ転送をサポートします。

INS命令は，DXが指すI/Oポートから ES:DIが指すメモリ領域へ複数バイト（またはワード）を転送します。この際，1バイト（または1ワード）転送するたびにDIは1（または2）ずつインクリメントされます（ただし，ディレクションフラグがセットされていればデク

リメントされます)。DXには影響を与えません。メモリ書き込みを行なうすべてのストリング命令と同様，デスティネーションセグメントとして指定したESを他のセグメントでオーバライドすることはできません。

OUTS命令は，DS:SIが指すメモリ領域からDXが指すI/Oポートへ複数バイト(またはワード)を転送します。この際，1バイト（または1ワード）転送するたびにSIは1（または2)ずつインクリメントされます（ただし，ディレクションフラグがセットされていればデクリメントされます)。DXには影響を与えません。セグメントオーバライドプレフィクスとして，DS以外のセグメントレジスタを指定することができます。次の例では，**INSB**で300hバイトのブロックをI/Oポート3000hからメモリへ転送し，**OUTSB**でそのブロックをI/Oポート3001hへ転送しています。

```
.
.
.
cld
mov   ax,@data
mov   ds,ax
mov   es,ax
mov   dx,3000h
mov   di,OFFSET Buffer
mov   cx,300h
rep   insb                  ; ポートからバッファへ300h分を転送
mov   dx,3001h
mov   si,OFFSET Buffer
mov   cx,300h
rep   outsb                 ; バッファからポートへ300h分を転送
.
.
.
```

8086命令の拡張版

80186は，次のような8086の拡張命令を持っています。

IMUL	**ROL**	**SAR**
PUSH	**ROR**	**SHL**
RCL	**SAL**	**SHR**
RCR		

即値の直接プッシュ

8086ではレジスタまたはメモリオペランドのみをプッシュできましたが，80186では次のように即値もプッシュできます。

```
push    19
```

即値を直接プッシュできれば，スタック経由で定数の引数を関数に引き渡す際に便利です。たとえば，

```
Average(5,2);
```

というCプログラムの関数 コールの8086コードは，次のようになります。

```
mov     ax,2
push    ax
mov     ax,5
push    ax
call    _Average
add     sp,4
```

一方，80186では次のようにプログラムがコンパクトになります。

```
push    2
push    5
call    _Average
add     sp,4
```

即値のシフトとローテート

8086では1ビットまたはCLで指定されたビット数だけシフトまたはローテートできましたが，80186では即値でシフトまたはローテートできます。

```
.
.
.
ror  ax,3
shl  dl,7
.
.
.
```

これによって，CLにシフトビット数をセットすることなく，複数ビット分のシフトを行なうことができます。次の例はAXを256倍する8086コードです。

```
  .
  .
  .
  mov  cl,8
  shl  ax,cl
  .
  .
  .
```

この処理は8086では次のようになります。

```
shl  ax,8
```

即値の乗算

8086では8または16ビットレジスタまたはメモリオペランドとALまたはAXをかけ,結果をAXまたはDX:AXにストアすることができました。80186では16ビット乗算の積を16ビットで表わすための2つの新しい乗算形式があります。

新しい乗算形式の1つは,16ビットレジスタと16ビット即値をかけ,結果を16ビットレジスタにストアする方法です。次の命令はDXに4をかけ,積をDXにストアします。

```
imul  dx,4
```

第1オペランドには任意の16ビット汎用レジスタが指定でき,一方の因数であると同時に積の格納先でもあります。第2オペランドは16ビット即値のもう1つの因数です。

新しい乗算方法のもう1つは,16ビットレジスタまたはメモリアドレスと16ビット即値をかけ,結果を指定の16ビットレジスタにストアする方法です。次の例はDXに600hをかけ,積をCXにストアします。

```
imul  cx,dx,600h
```

同様に,次の例は[BX+SI+1]で示されるアドレスにある16ビット値に3をかけ,積をAXにストアします。

```
imul  ax,[bx+si+1],3
```

このIMUL命令の第1オペランドは積の格納先です。このオペランドには任意の16ビット汎用レジスタが指定できます。第2オペランドは任意の16ビット汎用レジスタまたはメモリアドレスで,一方の因数です。第3オペランドは16ビット即値のもう1つの因数です。

少し考えれば,新しい乗算方法の最初の方法は,実は2番目の方法のサブセットになっていることがわかります。たとえば,

```
imul  si,10
```

という命令は,

```
imul  si,si,10
```

と結果的には同じです。

上記2種類のIMUL命令のマシンコードは共通です。ソースとデスティネーションの両方に同じレジスタを使う場合は,より単純な2オペランド形式のIMULを使う方が便利でしょう。

どちらの乗算方法でも，結果の値の16ビット内に入りきらない部分は失われます。上位ビットが失われた場合は，結果は符号付きと見なされキャリフラグまたはオーバフローフラグがセットされます。これらの乗算方法では，符号付きと符号なしの区別はありません。なぜなら，結果はわずか16ビット長であり，符号の有無にかかわらず16ビット値×16ビットの積の下位16ビットは常に同じだからです。したがって,新しい乗算方法としてはIMUL命令のみが使えます。

80286

80286は，1Mバイトのメモリ制限をなくし，メモリの保護モードと仮想モードをサポートした最初のiAPx86ファミリプロセッサです。80286は8086と80186のすべての命令の他，高度なメモリ管理を行なう多数の命令をサポートします。

80286は実モードと保護モードという2つの動作モードを持っています。実モードの80286は80186と同等で，完全に同じ命令セットを持ちます。IBM ATやNEC PC-9801VXなどの80286を搭載したコンピュータは，このモードでPC-DOS, MS-DOSおよびアプリケーションソフトを走らせています。

80286のメモリ管理機能は保護モードでのみ利用可能です。相互干渉することなく複数プログラムを同時に実行したり，1Mバイトを超えるメモリのアドレシングが行なえるのは保護モードだけです。保護モードは，80286を搭載したコンピュータがOS/2を走らせるモードでもあります。

80286の保護モード命令は次のとおりです。

CLTS	**LIDT**	**LMSW**
LGDT	**LLDT**	**LTR**

これらの命令は，オペレーティングシステムだけが使用し，アプリケーションプログラムは使用する必要がありません（また使用してはなりません）。これらの命令および80286の保護モードを使うには，専門的かつ複雑な知識が必要です。このマニュアルでは扱いません。

80286のフラグレジスタには，ネストタスクビットとI/O特権レベルフィールドという2つの新しいステータスが加えられています。保護モード命令と同様，これらの2つのビットはシステムソフトウェアだけが使い，アプリケーションプログラマには関係ありません。80286はまた，保護モード命令でのみ操作できる新しいレジスタをいくつか備えています。これらのレジスタは，タスクレジスタ，マシンステータスワードレジスタ，グローバルディスクリプタテーブルレジスタです。これらのレジスタもアプリケーションプログラムが使うことはありません。このマニュアルでは説明しません。

80286アセンブルモードにする

Turbo Assemblerでは，**.286**疑似命令によって非保護モードの80286命令のアセンブルを有効にします（**.286C**疑似命令も80286命令のアセンブルを有効にします。これは従来のアセンブラとの互換性を保つためです)。

80286はiAPx86ファミリの以前のすべてのプロセッサの命令セットをサポートするため，**.286**疑似命令を指定すれば8086と80186のすべての命令セットが自動的にサポートされることに注意してください。

80286の保護モード命令は，**.286P**疑似命令によって有効になります。**.286P**疑似命令は，80286の非保護モード命令も有効にします（**.286**疑似命令を指定したときと同様です)。

80286命令についての詳細は，**リファレンスガイド**の第3章を参照してください。

80286の保護モード命令の重要な点は，8086と80186がこれらの命令をまったく認識しないことです。したがって，保護モード命令を使うプログラムは8086または80186で走らせることはできません。80386は80286の保護モード命令と非保護モード命令をともにサポートします。

80386

80386プロセッサの出現は，マイクロコンピュータの歴史に残るできごとです。新しく拡張された命令，拡張された32ビットレジスタセット，最大4Gバイトのリニアセグメント，複数の8086プロセッサの同時エミュレート機能，高速でシフトとローテートを行なうバレルシフタ，ページメモリ，iAPx86ファミリでは最速のクロック速度（最速の処理速度）などのさまざまな特長を備えています。すでにおわかりのように，80386の全機能をサポートするためには8086, 80186, 80286のアセンブリ言語の拡張版が必要です。Turbo Assemblerは80386の拡張命令，モード，機能のすべてをサポートしています。

80386はきわめて高性能なプロセッサで，8086よりはるかに複雑です。したがって80386のプログラミングに関しては，そのすべてを説明することはできません。ここでは，Turbo Assemblerが持っている80386のサポート機能について説明します。

80386アセンブルモードの選択

80286と同じように，80386命令も特権命令と非特権命令の2種類にわけることができます。非特権命令は任意のプログラムから実行できますが，特権命令はカレント特権レベルが0（最高特権レベル）のプログラムだけが実行できます。80386の特権命令は80286の特権命令のスーパーセットで，80286の特権命令と同様，オペレーティングシステムだけが使用することを前提にしています。

80386の非特権命令は**.386**疑似命令で有効になります（**.386C**疑似命令を指定すると80386命令が有効になりますが，これは従来のアセンブラとの互換性を保つための命令です）。

80386は従来のiAPx86ファミリプロセッサのすべての命令セットをサポートします。そのため，**.386**疑似命令を指定すれば8086と80186のすべての命令および80286の非特権命令が自動的に有効になることに注意してください。

80386の特権命令のサポートは**.386P**疑似命令で有効になります。**.386P**疑似命令は，80386の非特権命令も有効にします（**.386**疑似命令を指定したときも同様です）。80386は80286のすべての特権命令をサポートするため，**.386P**疑似命令を指定すれば，すべての80286特権命令も自動的に有効になります。

80386のセグメント型

80386では，80286方式の64Kバイトセグメントと最大4G（ギガ）バイトのリニアセグメントの2種類のセグメント方式をサポートするために，**USE16**と**USE32**という2つの新しいセグメント型が必要となります。

64Kバイトのセグメント内の任意アドレスをポイントするためには，ベースレジスタまたはインデックスレジスタ（BX, SI, DI, BP）内にストアされた16ビットのオフセット値，またはダイレクトアドレシングオフセットとして使われる16ビットオフセット値が必要です。このようなセグメント方式は80286（と8086）の動作モードです。80386で最大64Kバイトのセグメントを扱う場合は，**USE16**というセグメント型を使います。次に例を示します。

```
       .386
        .
        .
        .
  DataSeg   SEGMENT  USE16
  Var1 DW   ?
  Ptr1 DW   Var1
  DataSeg   ENDS
        .
        .
        .
  Codeseg   SEGMENT  USE16
       ASSUME    CS:CodeSeg
       mov  ax,DataSeg
       mov  fs,ax
       ASSUME   FS:DataSeg
       mov  [Var1],0           ; Var1に0をセット
       mov  bx,[Ptr1]          ; 16ビットポインタをVar1にロード
       inc  WORD PTR fs:[bx]   ; Var1をインクリメント
        .
        .
        .
  CodeSeg   ENDS
        .
        .
        .
```

この例では，80386の2つの新しいエクストラセグメントの1つであるFSが使われていることに注意してください（もう1つはGSです）。

80386の8個の32ビット汎用レジスタのどれかにストアされたオフセットは，その値が0FFFFh(65535)を超えない限り，**USE16**セグメント内のアドレスを指定するために使える点にも注意してください。

4Gバイトのセグメント内の任意のアドレスを指すには,8個の32ビット汎用レジスタ内にストアした32ビットオフセット，またはダイレクトアドレシングオフセットとして使われる32ビットオフセットが必要となります。80386で最大4Gバイトのセグメントを扱う場合は，**USE32**というセグメント型を使います。次に例を示します。

```
      .386
      .
      .
      .
BigDataSeg   SEGMENT   USE32
Var1  DW   ?
Ptr1  DD   Var1
BigDataSeg     ENDS
      .
      .
      .
CodeSeg    SEGMENT  USE16
      ASSUME    CS:CodeSeg
      mov   ax,BigDataSeg
      mov   fs,ax
      ASSUME    FS:BigDataSeg
      mov   [Var1],0            ; Var1に0をセット
      mov   eax,[Ptr1]          ; 32ビットポインタをVar1にロード
      inc   WORD PTR fs:[eax]   ; Var1をインクリメント
      .
      .
      .
CodeSEg    ENDS
      .
      .
      .
```

この例では,ポインタレジスタとしてEAXを使っていることに注意してください。「80386のアドレシングモード」の節で説明しているように，80386では8個の32ビット汎用レジスタ（EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP）のどれでもベースレジスタまたはインデックスレジスタとして使用できます。

SMALL演算子と**LARGE**演算子は,オペランドのデフォルトオフセットサイズを変更するために使われます。**SMALL**は16ビットオフセット，**LARGE**は32ビットオフセットをそれぞれ指定します。次に例を示します。

```
    .386
    .
    .
    .
CodeSeg   SEGMENT   USE16
    ASSUME   CS:CodeSeg
    mov  ax,DataSeg
    mov  ds,ax
    ASSUME   DS:DataSeg
    mov  ax,[LARGE TestLoc]
    .
    .
    .
CodeSeg   ENDS
    .
    .
    .
DataSeg   SEGNMENT   USE32
TestLoc   DW   0
DataSeg   ENDS
    .
    .
    .
```

この例では，**LARGE**を使って*TestLoc*への参照を32ビットオフセットによる*TestLoc*への前方参照にしています（ただし**TestLoc**は**USE32**セグメント内にあります）。**LARGE**を指定しない場合は，*CodeSeg*という**USE16**セグメント内での前方参照に16ビットオフセットを使うため，エラーとなります。

SMALL演算子と**LARGE**演算子の機能は，オフセットサイズを16ビットまたは32ビットに切り替えるだけではありません。**SMALL**は8086の16ビットアドレシングモードでの命令アセンブルを指定します。この16ビットモードでは，最大64Kバイトのアドレシングが可能です。**LARGE**は80386の新しい32ビットアドレシングモード（「80386のアドレシングモード」の節を参照）での命令アセンブルを指定します。この32ビットモードでは，最大4Gバイトのアドレシングが可能です。

次に例を示します。

```
      .
      .
      .
      .386
      .
      .
      .
CodeSeg   SEGMENT   USE16
      .
      .
      .
      mov   ax,[small ebx+esi+1]
      .
      .
      .
CodeSeg   ENDS
      .
      .
      .
```

このプログラムは，次の命令としてアセンブルされます。

```
mov  ax,[bx+si+1]
```

この例では，**SMALL**を使って8086方式の16ビットアドレシングモードを指定しています。したがってアセンブルされたコードでは，EBXとESIのかわりにBXとSIが使われています。しかし次の例はアセンブルされません。

```
      .
      .
      .
      .386
      .
      .
      .
CodeSeg   SEGMENT   USE16
      .
      .
      .
      mov   ax,[small eax+ecx+1]
      .
      .
      .
CodeSeg   ENDS
      .
      .
      .
```

この例では，EAX＋ECX＋1が有効な16ビットアドレシングモードでないためにアセンブルされません（「80386のアドレシングモード」の節で説明しますが，EAX＋ECX＋1は32ビットメモリアドレシングモードとして有効です）。

SMALLと**LARGE**演算子の詳細および**USE16**型と**USE32**型のセグメントとこれらの演算子の関連については，この章の終わりの近くの「16ビットと32ビットの命令とセグメントの混在」の節で説明します。**USE32**型セグメントと**USE16**型セグメントの選択の仕方も，この節で説明します。

USE16または**USE32**セグメントの選択は，間接ジャンプのサイズに影響を与えます。これに関しては「32ビット命令ポインタ」の節で解説します。

セグメント定義で**USE32**も**USE16**も指定されていない場合は，80386のアセンブル時には**USE32**が指定されます。

簡略セグメント疑似命令と80386のセグメント型

.386疑似命令と簡略セグメント疑似命令の両方を使う場合は，セグメントには**DWORD**アラインメントが指定されます。なぜなら，80386を搭載したコンピュータが**ダブルワード**に配置したデータを最も速く処理できるためです。

簡略セグメント疑似命令を使う場合は，**.386**疑似命令が**.MODEL**疑似命令の前に指定されていれば**USE32**型セグメントが生成され，**.MODEL**疑似命令の後に指定されていれば**USE16**型セグメントが生成されます。次の例は32ビットのコードセグメントとデータセグメントを生成します。

```
.386
.
.
.
DOSSEG
.MODEL    LARGE
.DATA
.
.
.
.CODE
.
.
.
```

一方，次の例は16ビットのコードセグメントを生成します。

```
DOSSEG
.MODEL   LARGE
.386
.DATA
.
.
.
.CODE
.
.
.
```

48ビットのFWORDデータ型

USE32セグメントについての興味ある点は，**USE32**セグメント内アドレスへのfarポインタ（セグメント:オフセットのフルポインタ）のサイズが通常の4バイトではなく，6バイトになっていることです。この理由は，**USE32**セグメント内のオフセットのサイズが32ビット（=4バイト）のためです。たとえば**USE16**セグメントでは，*Buffer*という8000hバイトのバッファへのfarポインタは4バイトで表わされ，次のようにセットされます。

```
    .386
    .
    .
    .
DataSeg       SEGMENT  USE16
Buffer        DB   8000h DUP (?)
BufferPtr LABEL    DWORD
          DW  OFFSET Buffer
          DW  SEG Buffer
DataSeg        ENDS
    .
    .
    .
CodeSeg       SEGMENT  USE16
    ASSUME   CS:CodeSeg
    mov  ax,DataSeg
    mov  ds,ax
    ASSUME   DS:DataSeg
    les  bx,[BufferPtr]       ; ES:BXにBufferの16ビットセグメントと
                              ; 16ビットオフセットをロード
    .
    .
    .
CodeSeg        ENDS
    .
    .
    .
```

一方，USE32セグメントについては，*Buffer*へのfarポインタは6バイトで表わされ，次のようにセットされます。

```
     .386
     .
     .
     .
DataSeg      SEGMENT  USE32
Buffer       DB   8000h DUP (?)
BufferPtr LABEL   FWORD
          DD  OFFSET Buffer
          DW  SEG Buffer
DataSeg       ENDS
     .
     .
     .
CodeSeg      SEGMENT  USE32
     ASSUME   CS:CodeSeg
     mov  ax,DataSeg
     mov  ds,ax
     ASSUME   DS:DataSeg
     les  ebx,[BufferPtr]      ; ES:EBXにBufferの16ビットセグメントと
                               ; 32ビットオフセットをロード
     .
     .
     .
CodeSeg       ENDS
     .
     .
     .
```

このプログラムでは，FWORDデータ型が使われていることに注意してください。FWORDの値は6バイト長です。BYTE PTR, WORD PTR, DWORD PTRなどの演算子と同様に，次のようにFWORD PTR演算子を使うことができます。

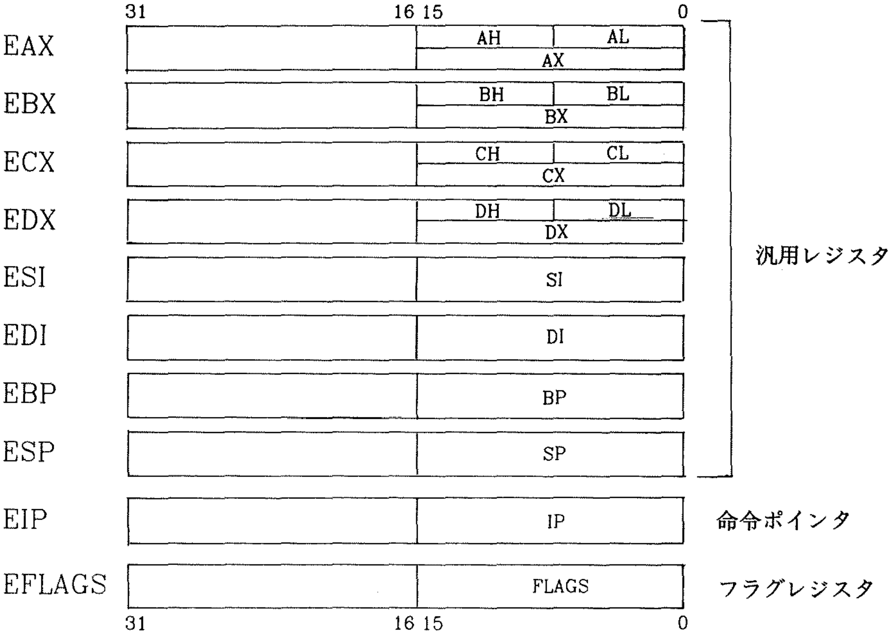
```
lgs   esi,FWORD PTR [BufferPtr]
```

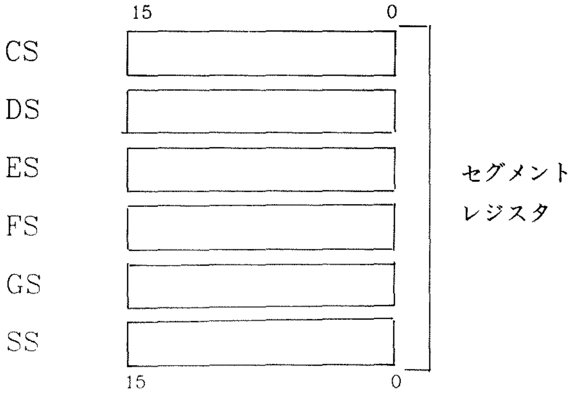
6バイト長の変数を定義するには，DF疑似命令を使います。次に例を示します。

```
    .386
    .
    .
    .
DataSeg    SEGMENT  USE32
FPtr DF    ?
DataSeg    ENDS
    .
    .
    .
CodeSeg    SEGMENT  USE32
    ASSUME   CS:CodeSeg
    mov  ax,DataSeg
    mov  ds,ax
    ASSUME   DS:DataSeg
    mov  eax,OFFSET DestinationFunction
    mov  DWORD PTR [FPtr],eax
    mov  ax,SEG DestinationFunction
    mov  WORD PTR [FPtr+4],ax
    jmp  [FPtr]
    .
    .
    .
CodeSeg       ENDS
    .
    .
    .
```

80386のレジスタ構成

80386には，8086の拡張版である32ビットサイズの汎用レジスタ，フラグレジスタ，命令ポインタが備っており，さらに2個の新しいセグメントレジスタが追加されています。図11.1に80386のレジスタセットを示します（拡張された部分は網かけで表わします）。





この他にも，80386にはいくつかの特殊レジスタがあります。これらの特殊レジスタには，新しいものと80286互換のものがありますが，どちらも特権命令でしか扱うことができません。80286では，これらのレジスタはシステムソフトウェアが使っています。このマニュアルでは説明しません。

では，80386の新しいレジスタについて見てみましょう。

32ビット汎用レジスタ

32ビット汎用レジスタは，EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESPです。これらのレジスタの下位16ビットは，従来の8086の16ビットレジスタセットと同じです。たとえばEAXレジスタの下位16ビットはAXレジスタ，下位8ビットはALレジスタです。つまりEAXレジスタは，32ビットEAXレジスタ，16ビットAXレジスタ，8ビットAHレジスタ，8ビットALレジスタの4種類のレジスタとして使いわけることができます。EBX, ECX, EDXの各レジスタも同様です。

80386の32ビット汎用レジスタは，16ビットや8ビットレジスタと同様に使うことができます。次に，EAXに1をストアし，EBXに0をセットし，EAXとEBXを加えるコード例を示します。

```
.
.
.
mov  eax,1
sub  ebx,ebx
add  ebx,eax
.
.
.
```

32ビット汎用レジスタは，16ビットレジスタが使える場所ならどこにでも使うことができます。

32ビットレジスタをアクセスする場合は，1つだけ不便なことがあります。それは，32ビットレジスタの上位16ビットを16ビットレジスタとして直接アクセスすることができないことです。たとえば，AXレジスタの上位8ビットはAHレジスタとして，ESIレジスタの下位16ビットはSIレジスタとしてそれぞれアクセスできます。ところがEAXレジスタの上位16ビットはアクセスできません。これでは，ワードとダブルワードの数値を同時に扱う場合などに不便です。しかし間接的にはアクセスすることができます。

32ビットレジスタの上位16ビットをアクセスするには，レジスタを16ビットだけどちらかの方向へローテートして，下位16ビットをアクセスして，もう一度16ビットだけローテートします。次の例は，AXに16ビット値をロードし，EDXをローテートして上位と下位の

ワードを入れ替え，もう一度EDXの上位と下位のワードを入れ替えています。

```
.
.
.
mov  ax,[Sample16BitValue]
ror  edx,16
mov  dx,ax
ror  edx,16
.
.
.
```

この例では，最初にAXにロードされた値が最終的にEDXの上位ワードに入ります。この手順は冗長にみえますが，実際の処理はそれほど遅くはありません。80386のバレルシフトでは，**ROR**命令の実行は3サイクルだけで済みます。

32ビットフラグレジスタ

80386のフラグレジスタの下位ワードは，80286のフラグレジスタと同一です。上位ワードには2つの新しいフラグが含まれます。1つは80386が仮想86モードで動作しているか否かを示すフラグで，もう1つはデバッグツールを作るときに使われるフラグです。通常これらのフラグは，アプリケーションソフトウェアが使うことはありません。

32ビット命令ポインタ

8086の命令ポインタのサイズが16ビットなのに対して，80386の命令ポインタのサイズは32ビットです。これによって最大4Gb（キガバイト）のコードセグメントをサポートすることが可能となります。

80386の拡張命令ポインタでは，メモリを介した間接ジャンプの指定方法が若干複雑になっています。次の命令は，16ビットのセグメントと32ビットのオフセットを使ったfar間接ジャンプを行ないます。

```
jmp   [FWORD PTR JumpVector]
```

次の例はどうでしょうか？

```
jmp   [DWORD PTR JumpVector]
```

この命令は32ビットのnear間接ジャンプまたは16ビットセグメントと16ビットオフセットのfar間接ジャンプのどちらでしょうか？　どちらの型のジャンプ命令も**DWORD**演算子

によって指定することができます。

次に**LARGE**と**SMALL**演算子の使用例を示します。

```
jmp    SMALL [DWORD PTR JumpVector]
```

この命令は，*JumpVector*にストアされている16ビットセグメントと16ビットオフセットのfar間接ジャンプとしてアセンブルされます。

```
jmp    LARGE [DWORD PTR JumpVector]
```

この命令は，CSの現在値と*JumpVector*にストアされている32ビットオフセットの組み合わせで指定される，near間接ジャンプとしてアセンブルされます。**SMALL**演算子を使うと，ジャンプ命令は**USE16**セグメントにあるものとみなされます。**USE16**セグメント内では，32ビット間接ジャンプのオペランドの構成要素は16ビットセグメントと16ビットオフセットです。**LARGE**演算子を使うと，ジャンプ命令は**USE32**セグメントにあるものとみなされます。**USE32**セグメント内では，32ビット間接ジャンプのオペランドの構成要素は32ビットオフセットだけです。

これらの例では，**SMALL**と**LARGE**演算子は大カッコの外側にあることに注意してください。**SMALL**と**LARGE**の位置は重要です。**SMALL**と**LARGE**をカッコの外側に置いた場合は，オフセットサイズ（この例ではジャンプサイズ）を指定します。カッコの内側に置いた場合は，アドレスサイズを指定します。次の命令は，32ビットのnearオフセットを使って*JumpVector*をポイントします。しかし*JumpVector*にストアされている値を32ビットのnearオフセットとして扱うか，16ビットセグメントと16ビットオフセットのペアとして扱うかは指定しません。

```
jmp    [LARGE DWORD PTR JumpVector]
```

したがって，この命令はジャンプの型を決定するという本来の機能を果たしていません。

SMALLも**LARGE**も，1つの式中の大カッコの外側でも内側でも使うことができます。たとえば次の命令は，*JumpVector*というダブルワード変数にストアされている16ビットセグメントと16ビットオフセットへのfar間接ジャンプを指定します。

```
jmp    SMALL [LARGE DWORD PTR JumpVector]
```

80386のセグメントレジスタ

80386には8086にはなかったFS，GSという2個の新しいセグメントレジスタがあります。これらのレジスタは特定の役割を持っていません。また暗黙にこれらのレジスタをアクセ

スする命令やアドレシングモードもありません。したがってFSとGSは特に必要となるわけではありませんが，同時に複数セグメント内のデータをアクセスする場合は便利です。

ESをストリング命令以外の命令に使う場合がそうであったように，FSとGSもセグメントオーバライドプレフィクスとして使います。次のプログラムは，オーバライドプレフィクスを明示的に指定した例です。

```
      .386
      .
      .
      .
TestSeg    SEGMENT  USE16
SCRATCH_LEN EQU     1000h
Scratch    DB    SCRATCH_LEN DUP(?)
TestSeg    ENDS
      .
      .
      .
CodeSeg    SEGMENT  USE16
      ASSUME    CS:CodeSeg
      mov  ax,TestSeg
      mov  fs,ax
      mov  bx,OFFSET Scratch
      mov  cx,SCRATCH_LEN
      mov  al,0
ClearScratch:
      mov  fs:[bx],al
      inc  bx
      loop ClearScratch
      .
      .
      .
CodeSeg    ENDS
      .
      .
      .
```

次のプログラムは，ASSUME疑似命令を使ってオーバライドプレフィクスを使った例です。

```
    .386
    .
    .
    .
TestSeg   SEGMENT  USE16
SCRATCH_LEN EQU    1000h
Scratch   DB   SCRATCH_LEN DUP(?)
TestSeg   ENDS
    .
    .
    .
CodeSeg   SEGMENT  USE16
    ASSUME    CS:CodeSeg
    mov  ax,TestSeg
    mov  gs,ax
    ASSUME    GS:TestSeg
    sub  bx,bx
    mov  cx,SCRATCH_LEN
    mov  al,0
ClearScratch:
    mov  [Scratch+bx],al
    inc  bx
    loop ClearScratch
    .
    .
    .
CodeSeg   ENDS
    .
    .
    .
```

この例では，**ASSUME GS:TestSeg**という疑似命令によって，*TestSeg*内の変数の名前によるアクセス（これに対してポインタレジスタによるアクセスもある）に対して，オーバライドプレフィクスを付けることが指定されます。したがって明示的にオーバライドプレフィクスを付ける必要はありません。しかし実行可能プログラム内ではオーバライドプレフィクスは依然として存在し，FSやGSレジスタでメモリをアクセスする命令のサイズはそのバイト分だけ長くなっています。結論として，FSやGSレジスタよりもDSレジスタをできるだけ使ってください（ストリング命令のデスティネーションにはESを使ってください）。

80386のアドレシングモード

80386は8086，80186，80286のすべてのアドレシングモードの他，パワフルな新しいアドレシングモードをサポートしています。80386の32ビット汎用レジスタは，どれもベースレジスタとして使えます。またSP以外の任意の32ビット汎用レジスタがインデックスレジス

タとして使えます。これに対して8086では，BXとBPだけがベースレジスタとして，SIとDIだけがインデックスレジスタとして使うことができました。

たとえばEDIに10000h, EAXに4が入っているとき，次の命令は80386では正常に動作します(DSで指されるセグメント内のオフセット10000h＋4＋2＝10006hのバイトをインクリメントします)。

```
inc   BYTE PTR [edi+eax+2]
```

次に80386の新しいアドレシングモードの他の例を示します。

```
 .
 .
 .
mov  ecx,[esp+4]
mov  ebx,[esp+8]
mov  WORD PTR [ecx+ebx],0
 .
 .
 .
```

80386の新しいアドレシングモードの応用例はこれだけではありません。インデックスレジスタをメモリアドレスの計算に使う場合は，インデックスレジスタの後に＊2，＊4，＊8を付けておくだけで，レジスタをそれぞれ2倍，4倍，8倍することができます。この機能は**インデックススケーリング**といいます。たとえば*DwordTable*というテーブルの9番目のダブルワードデータをEAXにロードするには，次のように行なうことができます。

```
 .
 .
 .
mov  ebx,8
mov  eax,[DwordTable+ebx*4]
 .
 .
 .
```

これは次のコードと同じです。

```
 .
 .
 .
mov  ebx,8
shl  ebx,2
mov  eax,[DwordTable+ebx]
shr  ebx,2
 .
 .
 .
```

インデックススケーリングは，ワード，ダブルワード，クオッドワード単位で配列をアクセスする際に特に便利です。次に，ワード配列の要素を昇順ソートするプログラム例を示します。

```
    .386
    .
    .
    .
CodeSeg   SEGMENT   USE32
    ASSUME    CS:CodeSeg
    .
    .
    .
;
; ワード配列を昇順ソートする
;
; 入力:
;    DS:EBX - ソートするワード配列の開始ポインタ
;    EDX    - 配列のワード要素数
;
; 破壊されるレジスタ:
;    AX,ECX,EDX,ESI,EDI
;
SortWordArray  PROC
    and  edx,edx
    jz   EndSortWordArray
    mov  esi,0                   ; 最初にすべての要素を0と比較
SortOnNextWord:
    dec  edx                     ; 比較する数をカウントダウン
    jz   EndSortWordArray
    mov  ecx,edx                 ; この要素と比較する要素数
    mov  edi,esi                 ; この要素を他の残りの要素と比較
CompareToAllRemainingWords:
    inc  edi                     ; 次の要素のインデックス
    mov  ax,[ebx+esi*2]
    cmp  ax,[ebx+edi*2]          ; カレントの要素 < 比較要素?
    jbe  NoSwap                  ; Yes...交換しない
    xchg ax,[ebx,+edi*2]         ; カレントの要素と比較要素を交換
    mov  [ebx+esi*2],ax
NoSwap:
    loop CompareToAllRemainingWords
    inc  esi                     ; 次の要素をポイントして他の残りの要素と比較
    jmp  SortOnNextWord
EndSortWordArray:
    ret
SortWordArray  ENDP
    .
    .
    .
CodeSeg  ENDS
    .
    .
    .
```

*SortWordArray*プロシージャは，ESIとEDIにストアされたカレント要素と比較要素の要素番号つまりインデックスを処理します。これらの値はポインタ（ワード配列に対しては2ずつカウントする）ではなく，単なるスカラの配列インデックスにすぎません（次のCの文で*n*が配列のインデックスであるのと同様です）。

```
i = Array[n];
```

*SortWordArray*プロシージャのキーポイントは，80386のインデックススケーリング機能によってメモリアドレシングフィールドとしてのインデックスを2倍して，インデックスをワード配列用のオフセットへ変換していることです。

1つのレジスタだけを使ってメモリをアドレシングする場合は，そのレジスタは常にベースレジスタとみなされます。2つのレジスタを使ってメモリをアドレシングする場合は，大カッコ内の左端レジスタはベースレジスタ，右端レジスタはインデックスレジスタとみなされます。ただし大カッコ内のこれらのレジスタのどちらかにスケーリングを適用した場合は，常にスケーリングされた方のレジスタがインデックスレジスタとみなされます。

特に指定のない限り，メモリアクセスの対象となるセグメントはベースレジスタがコントロールするため，どのレジスタがベースレジスタになるかは重要な問題です。EBPまたはESPをベースレジスタとするメモリアクセスでは，SSが指し示すセグメントを参照します。一方，EAX, EBX, ECX, EDX, ESI, EDIをベースレジスタとするメモリアクセスでは，DSが指し示すセグメントを参照します。次の例ではDSが指し示すセグメントを参照します。

```
mov  al,[eax]
xchg edx,[ebx+ebp]
shr  BYTE PTR [esi+esp+2],1
mov  [ebp*2+edx],ah
sub  cx,[esi+esi*2]
```

一方，次の例ではSSが指し示すセグメントを参照します。

```
rol  WORD PTR [ebp],1
dec  DWORD PTR [esp+4]
add  ax,[eax*2+esp]
mov  [ebp*2],edi
```

ベースレジスタによって選択されるデフォルトセグメントは，明示的なセグメントオーバライドプレフィクスによって，または**ASSUME**疑似命令によってオーバライドされます。次に例を示します。

```
    .386
    .
    .
    .
TestSeg   SEGMENT  USE32
Array1    DW   100h DUP(0)
TestSeg   ENDS
    .
    .
    .
CodeSeg  SEGMENT  USE16
    ASSUME   CS:CodeSeg
    mov  ax,TestSeg
    mov  fs,ax
    ASSUME   FS:TestSeg
    mov  dx,[ebx+Array1]      ; ASSUMEによって指定されるデフォルトオーバライド
    mov  esi,OFFSET Array1
    mov  cx,100h
IncLoop:
    inc  WORD PTR fs:[esi]    ; 明示的オーバライド
    inc  esi
    inc  esi
    loop IncLoop
    .
    .
    .
CodeSeg   ENDS
    .
    .
    .
```

80386の新しいアドレシングモードは，32ビットのメモリアドレシングレジスタでのみ有効です。16ビットレジスタは，8086で使う場合と同じ制限のもとでのみ使用することができます。次の**MOV**命令は，80386では使えません。

```
mov   ax,[cx+dx+10h]
```

また16ビットレジスタのインデックススケーリングは行なえません。また16ビットレジスタと32ビットレジスタを組み合わせてメモリアドレシングすることもできません。したがって，次の命令は使えません。

```
add  dx,[bx+eax]
```

80386の命令

次に，80386の新しい拡張命令を説明します。80386の命令の詳細は，**リファレンスガイド**の第3章を参照してください。

注意：8086, 80186, 80286 は，これから説明する拡張命令はサポートしません。したがってこれらの新しい拡張命令を使ったプログラムは，従来のプロセッサでは動きません。80386の新しい命令は次のとおりです。

BSF	**BTR**	**LFS**	**MOVZX**
BSR	**BTS**	**LGS**	**SET*xx***
BT	**CDQ**	**LSS**	**SHLD**
BTC	**CWDE**	**MOVSX**	**SHRD**

ビットのテスト

80386のビットテスト命令は，**BT, BTC, BTR, BTS**です。**BT**は基本的なビットテスト命令で，指定されたビットの状態をキャリフラグへコピーします。たとえば次のプログラムでは，EAXのビット3が0でなければ*Bit3Is1*へジャンプします。

```
       .
       .
       .
       bt   eax,3
       jc   Bit3Is1
       .
       .
       .
Bit3Is1:
       .
       .
       .
```

この例では，EAXに00000008hがストアされている場合は*Bit3Is1*へジャンプし，EAXに0FFFFFF7hがストアされている場合はジャンプしません。**BT**命令の第1オペランドは，テストの対象となる値がストアされた16/32ビット汎用レジスタまたはメモリアドレスです。**BT**の第2オペランドは，テストするビット番号（8ビット即値または16/32ビット汎用レジスタ）です。第2オペランドに汎用レジスタを指定する場合は，そのサイズは第1オペラン

ドと同じでなければなりません。

テストするビット番号はレジスタでも即値でも指定でき，テストの対象となるフィールドはメモリまたはレジスタにストアされた値です。次の例は，アドレス*Table*＋*ebx*＋*esi*＊*2*のワードのビット5の状態をキャリフラグにセットします。

```
.
.
.
mov   ax,5
bt    WORD PTR [Table+ebx+esi*2],ax
.
.
.
```

ビット番号は，最下位ビット0から最上位ビットへ向かってカウントされます。したがってALに80hがストアされているときは，ALのビット7がセットされます。

BTCは，キャリフラグにコピーされた値が指定されたビットの補数であることを除いて，**BT**と同様です。つまり指定されたビットが0ならキャリフラグは1に，指定されたビットが1ならキャリフラグは0になります。キャリフラグのステータスがテストされるビットの反転ビットでなければならない場合は，**BTC**を使えば**CMC**命令を使う手間を省くことができます。

BTRは，指定されたビットをキャリフラグへコピーした後にそのビットを0にセットすることを除いて，**BT**と同様です。同様に，**BTS**は指定されたビットをキャリフラグへコピーした後にそのビットを1にセットします。これらのビットテスト命令は，フラグのテストと設定を分割不能の1命令で実行する際に便利です(**分割不能**とは，フラグのテスト処理とフラグの設定処理の間に割り込みが発生しないことを表わします)。

ビットのスキャン

80386の**BSF**と**BSR**命令は，ワードまたはダブルワードのオペランド内の0でない最初のビットまたは最後のビットを見つけるのに便利です。**BSF**は，ソースオペランドのビット0（最下位ビット）から最上位ビットに向かって0でないビットのスキャニングを開始します。ソースオペランドの全ビットが0であった場合はゼロフラグがクリアされます。それ以外の場合はゼロフラグがセットされ，最初の0でないビットの番号がデスティネーションレジスタにストアされます。

次の例は，**BSF**を使って0でない最初の（最下位の）ビットを見つけてそのビット番号をDXレジスタにストアします。この例では，DXの0でない最初のビットの番号が2のため，CXに2がロードされます。

```
        .
        .
        .
        mov  dx,0001101010101100b
        bsf  cx,dx
        jnz  AllBitsAreZero
        shr  dx,cl
        .
        .
        .
AllBitsAreZero:
        .
        .
        .
```

この例では，CLをDXのシフト数として使っています。その結果，DXは0でない最下位ビットがビット0になるまで右へシフトされています。

BSFの第2オペランドはスキャンの対象となる16/32ビット汎用レジスタまたはメモリアドレスで，第1オペランドは0でない最初のビットの番号をストアする16/32ビット汎用レジスタです。両オペランドは同じサイズでなければなりません。

BSRは，ソースオペランドの最上位ビットから最下位ビットに向かって0でないビットのスキャニングを開始することを除いて，**BSF**と同様です。次の例は，*TestVar*のデータの0でない最上位ビットのインデックス（27）がEAXにストアします。

```
        .
        .
        .
TestVar   DD   0FFFFFF00h
        .
        .
        .
        bsr  eax,[TestVar]
        .
        .
        .
```

符号付き/符号なし拡張を伴うデータ転送

MOVZXと**MOVSX**命令は，8/16ビットの値を16/32ビットの汎用レジスタまたはメモリアドレスに転送します。この際，データサイズは自動的にデスティネーションのデータサイズに拡張されます。**MOVZX**は，デスティネーションの最上位ビットを0にします。**MOVSX**は，8/16ビットの値をデスティネーションのサイズに符号付きで拡張して16/32ビットの汎用レジスタまたはメモリアドレスにストアします。これら2つの命令は，標準の**MOV**命令と同じように使うことができます。　たとえば8086命令でDLにストアされている値をBX

へ転送する場合は，次のようになります。

```
.
.
.
mov   bl,dl
sub   bh,bh
.
.
.
```

一方，80386では次のようになります。

```
movzx   bx,dl
```

符号付き拡張は，8086ではもっと強力になります。**MOVSX**を使わずに*TestByte*という符号付きバイトメモリ変数をDXにコピーする場合は，次のようになります。

```
.
.
.
mov   al,[TestByte]
cbw
mov   dx,ax
.
.
.
```

MOVSXを使えば，次のように1命令で済みます。

```
movsx   dx,[TestByte]
```

MOVZXと**MOVSX**は，次のように8ビット値を32ビットレジスタに転送することもできます。

```
movsx   eax,al
```

DWORDまたはQWORDへのデータ変換

8086の**CBW**命令はALにストアされた符号付きバイト値を符号付きワード値に変換し，**CWD**命令はAXにストアされた符号付きワード値を符号付きダブルワード値に変換します。80386では，**CWDE**と**CDQ**という2つの符号付き数値変換命令が追加されています。これらの命令は，80386の32ビットレジスタをうまく使っています。

CWDEは，**CWD**と同様にAXにストアされた符号付きワード値を符号付きダブルワード値に変換します。両者の違いは，**CWD**はDX：AXに32ビットの結果をストアしますが，**CWDE**はEAXに32ビットの結果をストアすることです。CWDEの結果は，80386の32ビット命令ですぐに扱うことができます。

次の例では，結果の32ビット値はEAXにストアされます。

```
.
.
.
mov   ax,-1
cwde
.
.
.
```

CDQは，EAXにストアされた符号付きダブルワード値を符号付きクオッドワード（8バイト）値に変換してEDX：EAXにストアします。次に例を示します。

```
.
.
.
mov   eax,-7
cdq
.
.
.
```

この例は，64ビットのレジスタペアEDX：EAXに−7をストアします。この際，上位ダブルワード0FFFFFFFFhはEDXに，下位ダブルワード0FFFFFFF9h（−7）はEAXにストアします。

複数ワードにまたがるシフト

8086では，複数ワードにまたがるシフト（たとえば32ビット値を左へ4ビットすること）を簡単に行なうことはできません。各ワードを一度に1ビットしかシフトできないため，レジスタからキャリフラグを経由して他のレジスタへ1ビットずつ移動する方法がとられます。80386では，**SHRD**と**SHLD**命令によってこのような処理が簡単に行なえるようになっています。これらの命令は，2個のレジスタまたはレジスタとメモリにまたがるマルチビットシフトをサポートします。

たとえば，8086でDX：AXに格納された32ビット値を左へ（最上位ビットの方向へ）4ビットだけシフトする場合は，次のようなコードが必要です。

```
.
.
.
shl  ax,1
rcl  dx,1
shl  ax,1
rcl  dx,1
shl  ax,1
rcl  dx,1
shl  ax,1
rcl  dx,1
.
.
.
```

一方，80386では同じことを次の2命令で行なうことができます。

```
.
.
.
shld dx,ax,4
shl  ax,4
.
.
.
```

（もちろん，32ビット値がEAXレジスタに格納されていれば，下の命令でシフトすることができます。しかし上の例の目的は，8086命令のかわりに**SHLD**を使うことのメリットを示すことにあります)。

（リスト1行 R11_64.lst）

SHLDの第1オペランドはシフトの対象となる16/32ビットの汎用レジスタまたはメモリアドレス，第2オペランドは同じくシフトの対象となる16/32ビットの汎用レジスタ，第3オペランドはシフトするビット数です。第1オペランドと第2オペランドのサイズは一致していなければなりません。第3オペランドは即値またはCLです。CLを使った場合は，デスティネーションオペランドはCLの値で指定されたビット数だけシフトします。

SHRDは，右へ（最上位ビットから最下位ビットの方向へ）シフトすることを除いて，**SHLD**と同様です。次の例は，*TestQword*にストアされた64ビット値を7ビットだけ右へシフトします。

```
.
.
.
mov  cl,7
mov  eax,DWORD PTR [TestQword+4]
shrd DWORD PTR [TestQword],eax,cl
shr  eax,cl
mov  DWORD PTR [TestQword+4],eax
.
.
.
```

バイトの条件セット

条件テストと条件ジャンプの共通の応用例は，特定のステータスを表わすためにメモリを設定することです。たとえば，2つの変数が等しいか否か，ポインタがヌルか否か，キャリフラグが直前の命令によってセットされたか否かを示すためにフラグを設定したいことがあります。8086では，フラグを設定して条件テストの結果を表わすには（時間のかかるジャンプを含む）複数の命令が必要なため，このような処理を行なうのは困難です。しかし80386は，このようなテスト処理と設定処理の組み合わせを高速で実行するための**SET**という強力な命令を備えています。

たとえば，AXの最上位ビットが1のときにのみ*TestFlag*というメモリ変数をセットする場合は，8086では次のようにします。

```
    .
    .
    .
    mov  [TestFlag],0    ; MSBは1でないと仮定する
    test ah,80h
    jz   MSBNotSet
    mov  [TestFlag],1
MSBNotSet:
    .
    .
    .
```

80386では次のようになります。

```
.
.
.
test  ah,80h
setnz [TestFlag]
.
.
.
```

AHのビット7が1ならば*TestFlag*は1にセットされ，AHのビット7が0ならば*TestFlag*は0にリセットされます。

SET命令を使えば，ジャンプの条件をテストすることもできます。**SETNC**は，キャリフラグが0ならばデスティネーションを1にセットし，キャリフラグが1ならばデスティネーションを0にリセットします。**SETS**は，サインフラグが1ならばデスティネーションを1にセットし，サインフラグが0ならばデスティネーションを0にリセットします。**SET**命令のオペランドは，8ビットの汎用レジスタまたは8ビットのメモリ変数です。16ビットと32ビットのオペランドは指定できません。

SS, FS, GSの設定

8086では，**LDS**命令によってDSと汎用レジスタのどれかにメモリから1命令をロードすることができます。これによってfarポインタを効率よくセットアップすることができます。**LES**命令も同様の機能を持ちますが，DSでなくESにロードします。80386のfarポインタ設定のための命令は，**LSS, LFS, LGS**です。これらの命令は，それぞれSS, FS, GSセグメントレジスタに基づいてfarポインタを設定します。

次の例は，A000：0000にあるビデオビットマップへのfarポインタをGS：BXにロードします。

```
    .
    .
    .
DataSeg        SEGMENT  USE16
ScreenPointer LABEL     DWORD
    dw    0
    dw    0A000h
DataSeg   ENDS
    .
    .
    .
CodeSeg        SEGMENT  USE16
    ASSUME     CS:CodeSeg,DS:DataSeg
    mov   ax,DataSeg
    mov   ds,ax
    .
    .
    .
    lgs   bx,[ScreenPointer]
    .
    .
    .
CodeSeg        ENDS
    .
    .
    .
```

LDSと**LES**の場合と同様，**LSS，LFS，LGS**によってsmallまたはlargeのfarポインタを設定することができます。smallとlargeのfarポインタの詳細については，少し前の「48ビットのFWORDデータ型」の節を参照してください。

拡張命令

80386では,8086/80186/80286 の命令セットに新しいパワフルな命令が追加されたのみでなく，既存の命令が拡張されています。80386の拡張命令は次のとおりです。

CMPS	**JC**	**JNAE**	**JNLE**	**JPO**	**OUTS**
IMUL	**JCXZ**	**JNB**	**JNO**	**JS**	**POPA**
INS	**JE**	**JNBE**	**JNP**	**JZ**	**POPF**
IRET	**JG**	**JNC**	**JNS**	**LODS**	**PUSHA**
JA	**JGE**	**JNE**	**JNZ**	**LOOP**	**PUSHF**
JAE	**JL**	**JNG**	**JO**	**MOV**	**SCAS**
JB	**JLE**	**JNGE**	**JP**	**MOVS**	**STOS**
JBE	**JNA**	**JNL**	**JPE**		

さらに,80386ではニーモニックは同じでも32ビットのオペランドを指定できる命令が多数あります。

特殊形式のMOV命令

80386は，特殊形式の**MOV**命令をサポートします。この命令を使うと，特権レベル0(最高特権レベル)で動作しているプログラムは，32ビット汎用レジスタと80386の特殊レジスタとの間でデータ転送を行なうことができます。このような方法でアクセスできる80386のレジスタは次のとおりです。

CR0	DR0	DR3	TR6
CR2	DR1	DR6	TR7
CR3	DR2	DR7	

次のコードで，デバッグレジスタDR0にトラップするリニアアドレスをロードすることができます。

```
.
.
.
.386P
.
.
.
mov   eax,OFFSET FunctionEntry
mov   dr0,eax
.
.
.
```

また，次のコードでシステム制御フラグを制御レジスタCR0からEDXにロードすることもできます。

```
.
.
.
.386P
.
.
.
mov   edx,cr0
.
.
.
```

特殊形式の**MOV**命令は特権命令であるため，そのアセンブルを有効にするには**.386P**疑似命令を指定しなければならないことに注意してください。

特殊形式の**MOV**命令でアクセスできる80386の特殊レジスタは，通常，システムソフトウェアだけがアクセスでき，アプリケーションはアクセスすることはできません。

8086命令の32ビットバージョン

8086命令の多くは，80386の32ビットのアドレシングとオペランドで使えるように拡張されています。次の命令は，アドレス*EBP*＋*EAX*＊*8*＋*10h*にある32ビット変数から32ビットEBXレジスタの値を引きます。この例では，32ビットレジスタを使ってデスティネーションメモリアドレスを指しています。

```
sub   DWORD PTR [ebp+eax*8+10h],ebx
```

ほとんどの8086命令の32ビット拡張版では，ニーモニックはそのままです。命令が実行する32ビット処理は，オペランドによってまたは処理の対象となるセグメント型によって

示されます。ただし8086命令の中には，80386の32ビット処理をサポートするためにそのニーモニックが新しくなっているものもあります。次のこれらの命令について説明します。

LOOPとJCXZの32ビットバージョン

LOOP, LOOPE, LOOPNE, JCXZの各命令は，通常16ビットのCXレジスタで動作します。80386は，これらの命令の16ビットバージョンと32ビットバージョンを備えており，32ビットバージョンではCXではなくECXで動作します。

LOOP, LOOPE, LOOPNEの各命令では，セグメントの型（16ビットまたは32ビット）によって，CXまたはECXがループカウンタとして使われます。32ビットセグメント内においても常にCXをループ制御のレジスタとして使いたい場合は，これらの命令のワード形式である**LOOPW, LOOPWE, LOOPWNE**を使ってください。同様に，常にECXをループ制御のレジスタとして使いたい場合は，これらの命令のダブルワード形式である**LOOPD, LOOPDE, LOOPDNE**を使ってください。

LOOPDはECXをデクリメントし，結果が0でなければデスティネーションオフセットへジャンプします。たとえば，次のループは80000000h回実行されます。

```
    .
    .
    .
    mov  ecx,80000000h
LoopTop:
    loopd LoopTop
    .
    .
    .
```

LOOPDEはECXをデクリメントし，ゼロフラグが1でECXが0でない限りデスティネーションオフセットへジャンプします(**LOOPDZ**はこの命令のもう1つの形式で，意味は同じです)。同様に，**LOOPDNE**はECXをデクリメントし，ゼロフラグが0でECXが0でない限りデスティネーションオフセットへジャンプします（**LOOPDNZ**も同じです)。次のループは，DXが指すI/Oポートから読み込まれた値が09hになるかまたはポートが10000000h回チェックされるまで繰り返されます。

```
    .
    .
    .
    mov  ecx,10000000h
LoopTop:
    in   al,dx
    cmp  al,09h
    loopdne    LoopTop
    jnz  TimeOut
    .
    .
    .
TimeOut:
    .
    .
    .
```

ループ命令はステータスフラグに影響を与えないため，この例ではJNZの動作はLOOPDNE命令の結果ではなく比較命令の結果を反映することに注意してください。また80386には32ビット処理に適したJCXZ命令の拡張バージョンJECXZがあります。CXが0ならばJCXZでジャンプするのと同様に，ECXが0ならばJECXZでジャンプします。たとえば，次のループは32ビットのカウンタを扱うことができます。

```
    .
    .
    .
LoopTop:
    jecxz    LoopEnd
    .
    .
    .
    jmp  LoopTop
LoopEnd:
    .
    .
    .
```

ストリング命令の32ビットバージョン

80386では，すべてのストリング命令がバイト，ワード，ダブルワードの値を扱うことができます。ダブルワードバージョンのストリング命令の最後には，通常の*w*や*b*ではなく*d*が付きます。次に32ビットバージョンのストリング命令を示します。

CMPSD　**MOVSD**　**SCASD**
INSD　**OUTSD**　**STOSD**
LODSD

これらの各命令は一度に32ビットのデータを取り扱うことができ，1回の繰り返し処理で関連ポインタレジスタを4ずつインクリメント（またはデクリメント）します。たとえば，次のコードは**MOVSD**を使ってオフセット*DwordTable*から始まる2つのダブルワードをオフセット*Buffer*から始まる2つのダブルワードへコピーします。

```
.
.
.
cld
mov  si,OFFSET DwordTable
mov  di,OFFSET Buffer
mov  cx,2
rep  movsd
.
.
.
```

同じ処理を**MOVSB**を使って行なうと，次のようになります。

```
.
.
.
cld
mov  si,OFFSET DwordTable
mov  di,OFFSET Buffer
mov  cx,8
rep  movsb
.
.
.
```

ダブルワードストリング命令とワードストリング命令の関係は，ワードストリング命令とバイトストリング命令の関係に似ています。

IRETD

IRETDは**IRET**と同様です。EIPをポップした後でCSをダブルワードとしてポップし（上位ワードは捨てる），EFLAGSもダブルワードとしてポップします。

PUSHFDとPOPFD

PUSHFDは80386の32ビットのフルフラグレジスタをスタックにプッシュし，**POPFD**はそれをスタックからポップします。

それに対して，**PUSHF**と**POPF**はフラグレジスタの下位16ビットのみをそれぞれプッシュ，ポップします。

PUSHADとPOPAD

PUSHADは，32ビット汎用レジスタをEAX, ECX, EDX, EBX, ESP, EBP, ESI, EDIの順番でスタックにプッシュします。ESPに対してプッシュされる値は，**PUSHAD**実行時のESPの値です。**POPAD**は，32ビット汎用レジスタのうちの7個を**PUSHAD**とは逆順にポップします。したがってEDI, ESI, EBP, EBX, EDX, ECX, EAXは，**PUSHAD**でプッシュされ**POPAD**でポップされます。ESPだけは**POPAD**でポップされません。そのかわりESPは32だけインクリメントされます。その結果，**PUSHAD**でプッシュされた8個の32ビット汎用レジスタのブロックは捨てられ，ESPのプッシュ時の値は無視されます。

これに対して，**PUSHA**と**POPA**は8個の汎用レジスタの下位16ビットのみをそれぞれプッシュ，ポップします。

IMUL命令の32ビットバージョン

80386は，8086/80186/80286用の形式の**IMUL**命令に加えて，もっとも便利な**IMUL**命令を備えています。すべての汎用レジスタまたはメモリアドレスと任意の汎用レジスタをかけて，ソースレジスタの1つに結果をストアすることができます。オペランドの1つが定数であったり，アキュムレータがデスティネーションである必要はありません。次に例を示します。

```
imul  ebx,[edi*4+4]
```

この例では，EBXにメモリアドレスedi＊4＋4にストアされたダブルワードの値をかけて，結果をEBXにストアします。

すでにおわかりのように，この形式の**IMUL**命令の第1オペランドはデスティネーションレジスタで，任意の16/32ビット汎用レジスタが指定できます。第2オペランドは，任意の16/32ビット汎用レジスタまたはメモリアドレスが指定できます。これら2つのオペランドサイズは一致していなければなりません。符号付き数値としての結果のサイズがデスティ

ネーションのサイズを超える場合は，オーバフローフラグとキャリフラグが1にセットされます。

80386は，8086/80186/80286用形式の**IMUL**命令の拡張版を備え，32ビットオペランドをサポートします。たとえば，次の命令はECXと10000000hをかけて結果をEBPにストアします。

```
imul   ebp,ecx,10000000h
```

次の命令は，EBXとEAXをかけて結果をEDX：EAXにストアします。

```
imul   ebx
```

16ビットと32ビットの命令とセグメントの混在

通常は16ビットセグメント（**USE16**）だけで十分ですが，その場合でも32ビットレジスタを使って算術演算や論理演算を行なうことができます。

16ビットおよび32ビットのコードセグメントやデータセグメントを組み合わせて使うことができます。十分な知識があってオペレーティングシステムを作るのでない限り，32ビットのコードセグメントを使う必要はありません。特別な方法で32ビットのコードセグメントの実行に適したモードにプロセッサを切り替えない限り，DOSのもとで32ビットセグメントを使うことはできません。将来のオペレーティングシステムでは，32ビットコードセグメントを使えるようになっているでしょうが，現在はその必要はありません。

しかしプログラム内で32ビットデータセグメントを使えない理由や80386の32ビットレジスタのフラットアドレシングを利用しない理由はありません。

次に**USE16**や**USE32**セグメントのキーポイントを説明します。**USE16**セグメントのサイズは最大64Kバイトのため，**USE16**セグメント内のアドレスは16ビットで表現することができます。それに対して，**USE32**セグメントのサイズは最大4Gバイトのため，**USE32**セグメント内のアドレスは32ビットで表現することができます（**注意**：**USE32**コードセグメントは，**保護モードでのみ**使うことができます）。

64Kバイトよりも大きいセグメントが必要な場合は，**USE32**を使わなければなりません。

これに対して，**USE16**セグメントしか使ってはいけないということはありません。単純に考えて，つねに32ビットセグメントを使うようにできないのでしょうか？　この答えは，80386がどのようにワードやダブルワードのオペランドおよび16ビットと32ビットのオフセットをサポートしているかを見ればわかります。

80386は8086を改良して作られたプロセッサで，16ビットと32ビットの2種類のオペランドを区別するのに1ビットを使っています。8086のメモリアドレシングモードには(BX, SI, DI, BPを使った従来のアドレシングモード)の1セットしかなく，16ビットのオフセットのみをサポートしています。次の命令は，8ビットのサイズのオペランドを持ち，8086方式の16ビットアドレシングモードを使ってメモリをアドレシングします。

```
mov   al,[bx+1000h]
```

USE16コードセグメントでは，80386は通常8086と同じビットを使って8ビットオペランドと16ビットオペランドを切り替え，16ビットオフセットを使っています。ただし**USE16**セグメント内の命令は，その前にオペランドサイズプレフィクス（066h）を付けると，32ビットオペランドをサポートするようになります。この場合は，命令のサイズビットの指定によって，8ビットと16ビットのオペランドのみならず32ビットのオペランドも選択することが可能となります。

同様に**USE16**セグメント内の命令は，その前にアドレスサイズプレフィクス（067h）を付けると，80386の32ビットアドレシングモード（少し前の「80386のアドレシングモード」の節で説明したラージアドレス）が使えるようになります。

次に例を示します。

```
    .
    .
    .
    .386
    .
    .
    .
DataSeg    SEGMENT   USE16
TestLoc    DD    ?
DataSeg    ENDS
    .
    .
    .
CodeSeg        SEGMENT   USE16
    mov   ax,DataSeg
    mov   ds,ax
    ASSUME     DS:DataSeg
    db    66h
    mov   ax,WORD PTR [TestLoc]
    .
    .
    .
CodeSeg        ENDS
    .
    .
    .
```

この例では，オペランドサイズプレフィクスによって命令のオペランドサイズを32ビッ

トに変換しているため，*TestLoc*にある4バイトをEAXにロードします。

USE32コードセグメント内でこれと同じコードを使った場合は，命令は通常80386の32ビットアドレシングモードを使って8ビットまたは32ビットのオペランドをアクセスします。ただしオペランドサイズプレフィクスとアドレスサイズプレフィクスを使えば，**USE16**セグメント内で使うのと同じように，各命令を16ビットモード（ワードオペランドまたはスモールアドレスの一方または両方）で機能させることができます。

簡単にいうと，オペランドサイズプレフィクスとアドレスサイズプレフィクスを指定すれば，**USE16**コードセグメント内の命令は**USE32**セグメント内で使うのと同じように実行でき，**USE32**コードセグメント内の命令は**USE16**セグメント内で使うのと同じように実行することができます。

80386命令のオペランドサイズプレフィクスとアドレスサイズプレフィクスの使い方を学ぶ必要はありません。**USE32**セグメント内で16ビット機能を使うため，または**USE16**セグメント内で32ビット機能を使うために必要なプレフィクスは，Turbo Assemblerが自動的に生成します。たとえば**USE32**コードセグメント内で

```
mov   [bx],ax
```

という命令を使うと，Turbo Assemblerは自動的にオペランドサイズプレフィクスとアドレスサイズプレフィクスを命令に付けます。ここでは，サイズプレフィクスについてのみ説明しました。したがって16ビットまたは32ビットのセグメントサイズの選択のキーポイント，つまり生成するサイズプレフィクスの数を最小にすることの必要性を知ることができます。

たとえば，**USE16**セグメントを選択して，次のような命令で32ビットアドレシングモードを使ってダブルワードサイズのオペランドのみを参照した場合を考えてみましょう。

```
mov    eax,[edx+ecx*2+1]
```

この場合Turbo Assemblerは，コードサイズを膨張させ性能を落とす原因となるほとんどすべての命令に対して，オペランドサイズプレフィクスまたはアドレスサイズプレフィクスの一方または両方を生成する必要があります。**USE32**セグメントでは，同じコードでもサイズプレフィクスはまったく必要ありません。

セグメントサイズの選択プロセスは，実際よりも若干複雑にみえます。64Kバイトより大きなセグメントが必要な場合は，常に**USE32**セグメントを選択しなければなりません。64Kバイト以下のセグメントが必要な場合は，もし16ビットよりも32ビットのオペランドとアドレシングモードをより頻繁に使うならば**USE32**を選択し，そうでなければ**USE16**を選択しなければなりません。どちらのセグメント型がより効率が良いかは一概には決められませんが，2通りの方法でアセンブルしてみて，どちらのコードがよりコンパクトかをみるの

も1つの方法です。

これで, 前方参照アセンブルのためにLARGEやSMALL演算子を使わなければならない理由がわかりました。コードセグメントのUSEの型によってアドレス参照のデフォルトサイズが決まるため，コードセグメントのUSE型としては前方参照は同じサイズとみなされます。USE16コードセグメントからUSE32コードセグメントへの前方参照にはLARGEを使わなければなりません。また，USE32コードセグメントからUSE16コードセグメントへの前方参照に対して16ビットアドレシングモードを使うためには, SMALLを使わなければなりません。

80386の関数例

では，80386のサンプルプログラムを紹介します。本来は80386の完全なプログラムを載せるのが望ましいのですが, 現時点では広く普及している80386専用のオペレーティングシステムがないため，それはできません。またメモリリクエスト，キー押下の検出，画面表示，プログラム終了などの標準的な処理方法もありません。とりあえず，80386アセンブラコードで書かれた1つの関数を例にとります。

このサンプル関数*CalcPrimes*は, USE32の持つ膨大なメモリ空間を利用することによって，与えられた範囲内の素数を直接計算します。この関数は，2から指定した最大素数までの範囲内のすべての倍数を単純計算し, 1つの大きなテーブル内の素数でない倍数をすべてマークします。8086では，この方法はサイズが1セグメントのサイズつまり64Kバイト以下の配列にのみ有効で，使用するメモリの総容量が8086プロセッサが扱える最大メモリサイズである1Mバイトに達すると，処理不能となります。

これに対して，80386のUSE32セグメントと32ビットレジスタを使えば，4Gバイトに近いサイズのテーブルを簡単に扱うことができます。事実，80386ではページメモリの手法を使ってテラバイト(1000Gバイト)単位のメモリを扱うことさえ可能です。もちろん，そのような膨大な数の素数をチェックするには非常に長い時間を要します。8086や80286とは異なり, 80386のメモリアドレシングアーキテクチャは膨大なメモリを必要とするプログラムに対して何ら制限を与えることがありません。

次に*CalcPrimes*のプログラムを示します。

```
;
; 0～MAX_PRIME(範囲内に含む)の範囲にある
; すべての素数を計算する80386サンプルプログラム
;
; 入力:なし
;
; 出力:
;    ES:EAX - PrimeFlagsへのポインタ
;                PrimeFlagsの内容
;                  素数の場合    ...PrimeFlagsの対応オフセット = 1
;                  素数でない場合...PrimeFlagsの対応オフセット = 0
;
; 破壊されるレジスタ:
;    EAX, EBX
;
; PC Magazine Vol.7, No.2
; "Environments,"(by Charles Petzold)で紹介されたアルゴリズムに基づく
;
    .386

MAX_PRIME EQU 1000000          ; 素数か否かをチェックする最大値

DataSeg        SEGMENT  USE32
PrimeFlags     DB   (MAX_PRIME + 1) DUP (?)
DataSeg        ENDS

CodeSeg   SEGMENT  USE32
    ASSUME    CS:CodeSeg
CalcPrimes    PROC
    push ds                    ; 呼び出し側DSを退避
    mov   ax,DataSeg
    mov   ds,ax
    ASSUME   DS:DataSeg
    mov   es,ax
    ASSUME   ES:DataSeg
;
; 指定範囲内のすべての数を素数と仮定する
;
    mov   al,1
    mov   edi,OFFSET PrimeFlags
    mov   ecx,MAX_PRIME+1
    cld
    rep   stosb
;
; 2～MAX_PRIMEの範囲のすべての数に対する
; すべての倍数(MAX_PRIMES以下で1を除く)を計算して
; すべての非素数を取り除く
;
    mov   eax,2                ; 初期値は2
                               ; 0と1は素数であり,倍数の除去処理には使わない
PrimeLoop:
    mov   ebx,eax              ; すべての倍数を計算するための基数
MultipleLoop:
    add   ebx,eax              ; 次の倍数を計算
    cmp   ebx,MAX_PRIME        ; この数のすべての倍数をチェックしたか?
    ja    CheckNextBaseValue   ; Yes...次の数へ
```

```
    mov  [PrimeFlags+ebx],0   ; 何かの倍数であり,素数でない
    jmp  MultipleLoop         ; 次の倍数を取り除く
CheckNextBaseValue:
    inc  eax                  ; 次の基数(すべての倍数を計算するための
                              ; 次の数)をポイント
    cmp  eax,MAX_PRIME        ; すべての倍数を取り除いたか?
    jb   PrimeLoop            ; No...次の倍数セットをチェック
;
; 素数と非素数の一覧テーブルへのポインタをES:EAXに返す
;
    mov  eax,OFFSET PrimeFlags
    pop  ds                   ; 呼び出し側DSを復帰
    ret
CalcPrimes   ENDP
CodeSeg      ENDS
        END
```

80386では，32ビットの整数や1000000バイト長の配列を簡単に扱えることがわかります。事実，この関数の全コードはわずか20バイトです。*CalcPrimes*は，結果として*PrimeFlags*というテーブルを指すラージ型のfarポインタを返します。このテーブルでは，数字に対応するオフセットに1または0がセットされています。数字が素数なら1が，そうでなければ0がセットされます。たとえば，3は素数で4は素数でないため，*PrimeFlags*＋*3*には1が，*PrimeFlags*＋*4*には0がストアされます。

*PrimeFlags*の長さおよび素数か否かがチェックされる最大値は，*MAX_PRIME*というラベルでイコール定義されています。呼び出し側ルーチンがチェックする最大値(=テーブル長−1)とともに任意サイズのテーブルアドレスを*CalcPrimes*へ引数として引き渡すのがより実用的です。呼び出し側プログラムは必要に応じて*CalcPrimes*に素数計算を依頼することができ，テーブルサイズが変わるたびにアセンブルし直す必要はありません。この例では，**USE32**の使い方を主に示すために*PrimeFlags*というローカル変数を使っています。

次に，テーブルとテーブル長を引数として引き渡してコールする*CalcPrimes*を示します。

```
;
; 0～MAX_PRIME(範囲内に含む)の範囲にある
; すべての素数を計算する80386サンプルプログラム
;
; 入力: (ラージ型farコールを想定....6バイトのリターンアドレスをスタックに
;        プッシュする)
;
;    エントリでのESP+06h (最後のプッシュされる引数)
;       = 素数か否かがチェックされる最大値のダブルワード値
;
;    エントリでのESP+0Ah (最初のプッシュされる引数)
;       = テーブルを指すラージ型のfar(6バイトオフセット)ポインタ
;            テーブルの内容
;               素数の場合      ...テーブルの対応オフセット = 1
;               素数でない場合...テーブルの対応オフセット = 0
;            テーブルの長さは最低[ESP+06h]+1バイト,[ESP+06h]には他の引数がある
```

```
; 出力: なし
;
; 破壊されるレジスタ:
;     EAX, EBX, EDX, EDI
;
; PC Magazine Vol.7, No.2
; "Environments,"(by Charles Petzold)で紹介されたアルゴリズムに基づく
;
    .386

CodeSeg   SEGMENT  USE32
    ASSUME   CS:CodeSeg
CalcPrimes   PROC FAR
    push es                    ; 呼び出し側ESを退避
    push fs                    ; 呼び出し側FSを退避
;
; 引数を取得する
;
    mov   ecx,[esp+4+06H]
    mov   edx,[esp+4+0aH]
;
; 指定範囲内のすべての数を素数と仮定する
;
    push fs
    pop   es                   ; ESはテーブルのセグメントをポイント
    mov   al,1
    mov   edi,edx
    cld
    push ecx                   ; チェックする最大値を退避
    inc   ecx                  ; 最大値を設定(範囲内に含む)
    rep   stosb
    pop   ecx                  ; チェックする最大値を復帰
;
; 2～最大値の範囲のすべての数に対するすべての倍数(最大値以下で1を除く)を
; 計算してすべての非素数を取り除く
;
    mov   eax,2                ; 初期値は2
                               ; 0と1は素数であり，倍数の除去処理には使わない
PrimeLoop:
    mov   ebx,eax              ; すべての倍数を計算するための基数
MultipleLoop:
    add   ebx,eax              ; 次の倍数を計算
    cmp   ebx,ecx              ; この数のすべての倍数をチェックしたか?
    ja    CheckNextBaseValue   ; Yes...次の数へ
    mov   BYTE PTR fs:[edx+ebx],0 ; 何かの倍数であり,素数でない
    jmp   MultipleLoop         ; 次の倍数を取り除く
CheckNextBaseValue:
    inc   eax                  ; 次の基数(すべての倍数を計算するための
                               ; 次の数)をポイント
    cmp   eax,ecx              ; すべての倍数を取り除いたか?
    jb    PrimeLoop            ; No...次の倍数セットをチェック
    pop   fs                   ; 呼び出し側FSを復帰
    pop   es                   ; 呼び出し側ESを復帰
    ret
CalcPrimes    ENDP
CodeSeg       ENDS
        END
```

80287

80287数値演算コプロセッサの命令セットは，ただ1つの命令を除いて8087の命令セットと同じです。この命令は，80287を保護モードにするための**FSETPM**命令です。80287の保護モードは，80286プロセッサの保護モードに対応するものです(80286と80287はよく組み合わせて使われます)。もちろん，8087は**FSETPM**をサポートしないため，この命令を使ったプログラムを8087で動かすことはできません。

Turbo Assemblerでは，**.287**疑似命令を指定すれば80287のアセンブルモードが有効になります。80287の命令セットの詳細は，**リファレンスガイド**の第3章を参照してください。

80387

80387数値演算コプロセッサの命令セットは，8087または80287の命令セットのサブセットです。80387の新しい命令を次に示します。

FCOS	**FSINCOS**	**FUCOMP**
FPREM1	**FUCOM**	**FUCOMPP**
FSIN		

FUCOMは，ST(0)と他の80387レジスタとの無順序比較を行ないます。この命令は**FCOM**と同様です。ただしオペランドの1つがNaNの場合は，**FCOM**のように無効処理例外を生成するのではなく，結果のステータスを無順序にセットする点が異なっています。**FUCOMP**は，無順序比較を行ない，80387のスタックをポップします。**FUCOMPP**は，無順序比較を行ない，スタックを2回ポップします。

FCOSと**FSIN**は，それぞれST(0)レジスタのコサインとサインを計算します。

FPREM1は，ST(0)をST(1)で割ったときのIEEE互換の余りを計算します。

これらの命令のどれかを使うプログラムは，8087や80287では動かないことを忘れないでください。また80387は実モードと保護モードを同様に扱うため，80287では**FSETPM**命令は無視されます。

Turbo Assemblerでは，**.387**疑似命令を指定すれば80387のアセンブルモードになります。80387の命令セットの詳細は，**リファレンスガイド**の第3章を参照してください。

第12章

Turbo AssemblerのIdealモード

MASMを使ってプログラムを作るプログラマには，この章がこのマニュアルの中でもっとも重要になります。Turbo Assemblerは，MASMの構文とほぼ完全な互換性を保っているだけではなく，Idealモードと呼ばれるMASM疑似命令を備えています。このIdealモードによって，アセンブリ言語のプログラミングが円滑に行なえるようになっています。

Idealモードを使えば，ソーステキスト中の式または命令のオペランドの働きが明確に理解できます。またMASMのさまざまな癖や特徴を憶えなくとも，明瞭かつ一貫性のあるプログラムを書くことが可能となります。

Idealモードでは，MASMのほとんどすべてのキーワード，演算子，文法を使います。したがって，さまざまな新しい規則やキーワードを憶えなくとも，Idealモードの機能を1つずつ体験することができます。Idealモードのすべての機能は，MASMの既存機能の拡張版または改良版ということができます。

この章ではIdealモードの機能を解説し，Idealモードの新しい構文規則を使えば，いかに時間と労力が節約できるかを説明します。またIdealモードのすべての新しい機能の詳細，およびIdealモードとMASMモードにおける構文の違いについても解説します。

Idealモードとは何か？

Turbo AssemblerのIdealモードでは，式と命令のオペランドに新しい構文を使います。この新しい構文は，MASMの既存の構文とそうかけ離れたものではありません。Idealモードの構文は，プログラマにもTurbo Assembler自身にもわかりやすい形式になっており，MASMの演算子やキーワードをより単純明快にしたものと考えることができます。

Idealモードでは，式に対して厳密な型チェックが行なわれます。この型チェックによって，間違った型の値をレジスタや変数へ代入することによって発生するエラーや，ソーステキスト内では一見正しいがアセンブル時にうまく処理できないプログラム構造に起因するエラーなどを減らすことができます。またIdealモードを使えば，値や式であれこれ悩む必要はなくなり，論理的で美しいプログラムを書くことができます。

厳密な型チェックがあるため，Idealモードの式はより理解しやすく，予期せぬ結果が発生する確率が低くなっています。つまりIdealモードの監視機能の元では，これまで他の章で説明してきたMASMにおける多くの問題はすべて解決します。

Idealモードは，初心者にとってもエキスパートにとってもプログラミングの助けとなる多数の機能を備えています。これらの機能の一部を次に紹介します。

■複数の構造体で重複できるメンバ名

■複雑な**HIGH**と**LOW**演算子

■予測可能な**EQU**定義の処理

■グループ化したデータセグメントの適切な扱い

■より一貫性のある疑似命令

■大カッコを有効利用した式

なぜIdealモードを使うのか？

Turbo AssemblerのIdealモードを使わなければならない理由はたくさんあります。もしアセンブリ言語を知っていれば，必要な効果を持つIdealモードの式と文を容易に作ることができます。所定の命令が得られるまで，いろいろと試す必要はありません。経験のあるアセンブリ言語プログラマなら，Idealモードの機能を使ってネスト可能な構造体や共用体などの拡張機能を利用した複雑なプログラムを作ることができます。

Idealモードでは，構文が明瞭な分だけファイルのアセンブル速度がMASMモードより30%向上します。プロジェクトとファイルのサイズが大きくなるほど，Idealモードに切り換えた方がアセンブル時間を短縮することができます。

Idealモードの強力な型チェック規則によって，従来なら実行時またはデバッグ時でないと発見できないエラーを見つけることができます。これは高級言語のコンパイラが不明なプログラム構造やデータサイズの不一致などを指摘する方法に似ています。

Idealモードでは式によって異なった構文を使いますが，MASMモードでもIdealモードでも同じようにアセンブルできるプログラムを書くことができます。また1つのソースファイル内で，必要に応じて何回でもMASMモードとIdealモードを切り換えることができます。この機能は，Idealモードでチェックしたり MASMモードで書かれた既存プログラムを変換するときに特に役に立ちます。また，プログラムの他の部分はMASMとの完全な互換性を保ちつつ，ソースファイルに新規コードを追加することもできます。

Idealモードの開始と終了

IdealモードまたはMASMモードを有効にするには，それぞれ**IDEAL**疑似命令と**MASM**疑似命令を使います。Turbo Assemblerは，常にMASMモードでソースファイルのアセンブルを開始します。Idealモードへ切り換えるには，Idealモードの機能を使う前にソースファイルの中に**IDEAL**疑似命令を入れておきます。**IDEAL**疑似命令から次の**MASM**疑似命令までの間，すべての命令はこの章の説明のように機能します。MASMモードとIdealモードの切り換えは，いつでもどこでも必要なだけ何回でも行なうことができます。次に例を示します。

```
DATA SEGMENT             ; MASMモードで開始
abc  LABEL BYTE          ; abcはxyzをバイトとしてアドレシングする
xyz  DW    0             ; xyzにワードを定義
DATA ENDS                ; データセグメントの終わり

     IDEAL               ; Idealモードへ切り換える

SEGMENT CODE             ; セグメントキーワード
PROC MyProc              ; プロシージャキーワード
        .
        .                ; Idealモードのプログラム
        .
ENDP MyProc              ; MyProcラベルの繰り返しは任意
ENDS                     ; セグメント名の繰り返しは必要ない

     MASM                ; MASMモードへ戻る
CODE SEGMENT             ; セグメントキーワードの前に名前を付ける
Func2 PROC               ; プロシージャキーワードの前に名前を付ける
        .
        .                ; MASMモードのプログラム
        .
     IDEAL               ; Idealモードへ再び切り換える
        .
        .                ; Idealモードのプログラム
        .
     MASM                ; MASMモードへ戻る

Func2 ENDP               ; キーワードの前に名前を付ける
CODE ENDS                ; キーワードの前に名前を付ける
```

この例を見てもわかるように，Idealモードでは，**PROC**や**SEGMENT**などの疑似命令のキーワードが識別シンボル名の前にきます（MASMモードではその逆です）。また**ENDP**の後のプロシージャ名や**ENDS**の後のセグメント名は省略可能です。ただし名前を入れたほうが，終了するセグメント名やプロシージャ名が認識できてプログラムが明瞭になります。複数のセグメントやプロシージャをネストしているプログラムでは，シンボル名を付

けておいたほうが便利です。ただし必要がなければ省略できます。

MASMモードとIdealモードの違い

この節では，IdealモードとMASMモードの違いを説明します。MASMモードをすでに知っている場合は，既存プログラムの部分的な独立機能をIdealモードに切り換えて試すことができます。その際，Idealモードにする部分を**IDEAL**と**MASM**の疑似命令で囲むことを忘れないでください。このようなやり方で順次Idealモードを学ぶようにすれば，Idealモードの特殊機能を使うためにそれぞれの命令を書き直すことなく，現在のプログラムをアセンブルすることができます。もちろん最終的にはIdealモード専用のプログラムを作ることもできます。またはMASMモードとIdealモードのモジュールを混在させ，両者を比較することもできます。

Idealモードのトークン

Turbo Assemblerは，テキストを「トークン」と呼ばれるワードまたはシンボルに分割することによってプログラムを解読します。トークンの例は，*VALUE, NAME, AGE*などのラベルを含むシンボル名，数字，式の一部，＋－＊／などの算術演算子です。

2種類のトークン（シンボルトークンと浮動小数点トークン）は，Idealモードでは若干異なる形式を持ちます。次に説明するように，これらのIdealモードの形式を知れば，MASM構文のあいまいさが理解できます。

シンボルトークン

Idealモードでは，シンボル名にピリオド（.）を使うことはできません。ピリオドは構造体メンバまたは浮動小数点数に対してのみ使うことができます。

構造体や共用体のメンバ（フィールドともいう）は，グローバルシンボルとして定義することはできません。言い換えると，プログラム内のどこからでもアクセスできるように定義することはできません。構造体や共用体のメンバは，自身が所属するデータ構造内でのみ有効です。したがって複数のデータ構造で同じメンバ名を使うことができます。またデータ構造の外で同じメンバ名を他の目的に使うこともできます。次に例を示します。

```
Pennies  DW 0
STRUC Heaven
Dimes    DW ?
Nickels  DW ?
Pennies  DW ?           ; 二重定義とはならない
ENDS
Take  Heaven <>
```

この例は,3つのフィールドを持つ変数を定義して値を*Heaven*という名の構造体の*Dimes, Nickels, Pennies*というメンバにストアする方法を示します。*Dimes*と*Nickels*の2つのフィールドは構造体に固有の名前ですが, *Pennies*は以前にも使われた名前です。1回目は構造体の外側で, 2回目は構造体の内側で使われています。

この例は, *Pennies*というシンボル名が構造体の外側と内側で使われていても, 何も支障がないことを示しています。これはMASMモードではできません。

*Heaven*の外側の*Pennies*という変数は, *Heaven*のメンバの*Pennies*とは区別されます。つまり構造体の内側にある重複シンボル名を参照するには, 構造体名, ピリオド, メンバ名の3要素が必要となります。この例では, *Take.Pennies*は*Heaven*内の*Pennies*フィールドのオフセットを表わします。*Pennies*は構造体の外側の変数のオフセットを表わします。

重複メンバ名

Idealモードでは, 同じメンバ名を異なる構造体で使うことができます。メンバの型は同じであっても異なっていてもかまいません。次の2つの構造体は, 同じ場所に同じ型の*Size*というフィールドを持っています。また異なった場所に異なる型の*Amount*というフィールドを持っています。

```
STRUC SomeStuff
Size      DW    ?
Flag      DB    ?
Amount    DW    ?
ENDS

STRUC OtherStuff
Size      DW    ?        ; 二重定義とはならない
Amount    DB    ?        ; 二重定義とはならない
ENDS
```

浮動小数点トークン

Idealモードでは，浮動小数点10進数は常にピリオド（.）を含んでいます。次に例を示します。

```
FP    DT    1.0e7      ; Idealモードの浮動小数点数
```

この例は，FPという名の10バイト浮動小数点数（=1.0e7）を定義します。MASMモードでは次のような，ややわかりにくい形式になります。

```
FP    DT    1E7        ; MASMモードの浮動小数点数
```

この形式は，プログラムのはじめの方で**.RADIX 16**という疑似命令を使いさえしなければ特に都合が悪いものではありません（**.RADIX 16**は，デフォルトの基数を10進数から16進数に変更します)。もしMASMモードでこの疑似命令を使うと，この浮動小数点数は16進数の01E7hとアセンブルされてしまいます。したがってIdealモードでは浮動小数点数に必ず小数点を付けることによって，16進数との混同を防いでいます。

文字列と数値のイコール定義（EQU疑似命令と＝疑似命令）

Idealモードでは**EQU**定義（イコール定義）は常に文字列としてのみ扱われます。一方MASMモードではイコール定義は文字列としても数値としても扱われます。次に例を示します。

```
; いくつかのイコール定義を宣言
A     =     4
B     =     5
C     EQU   B + A
B     =     6

; 変数を宣言
V     DW    C             ; MASMモードでは9, Idealモードでは10
```

MASMモードでは$B+A$を**EQU**定義の処理時に評価するため，Cは5（$A=4$, $B=5$）となります。一方，Idealモードではその評価法方が異なります。Idealモードでは，**EQU**定義以降の項目はすべて文字列形式（この場合は$B+A$）でストアし，その後Cを見つけたときにこの文字列に値を代入します。したがってこの例では，変数Vの宣言時に初めて式が評価されます。またBは6に再定義されているため，変数Vの値は10（6+4）となります。

Idealモードでは，**EQU**定義は常に文字列を定義して等号（=）は常に計算済み（評価済み）の式を定義します。等号（=）が即座に式評価を行なうのに対して，**EQU**が定数名のあ

る場所まで式評価を遅らせると考えれば，この規則は覚えやすいでしょう。このような評価方法をそれぞれ「早い」連結および「遅い」連結と呼ぶ場合もあります。

式とオペランド

Idealモードと MASMモードの式のもっとも大きな違いは，大カッコの役割です。Idealモードでは，大カッコに囲まれた項目は常に内容を表わします。暗黙的な項目や機能は伴いません。したがって，Idealモードでは多くの標準的なMASM構文を使うことは許されません。

大カッコ演算子

Idealモードでは，項目の内容を取得するためには必ず大カッコを使わなければなりません。たとえば，

```
mov   ax,wordptr
```

という命令を使うと，警告メッセージが表示されます。レジスタ（AX）にポインタ（*wordptr*）の内容をロードする場合は，次の形式をとらなければなりません。

```
mov   ax,[wordptr]
```

この命令が（DSが指すカレントデータセグメント内の）*wrodptr*によって示されるアドレスの内容をAXにロードすることは明らかです。

あるセグメント内のシンボルのオフセット自身をロードする場合は，次のように**OFFSET**演算子を明示的に指定しなければなりません。

```
mov   ax,OFFSET wordptr
```

オペランドの例

次にMASMモードにおける（典型的な）大カッコの混乱した使用例を見てみましょう。また，それらの例をIdealモードの正しくてわかりやすい形式と比較してみましょう。Idealモードの明確な大カッコの使用法に準拠すれば，オペランドの使い方を完全に明確に記憶することができます。

```
mov  ax,[bx][si]            ; MASMモード
```

この命令はIdealモードでは構文エラーとなります。もし大カッコがメモリの内容を指定するのであれば，この命令はBXが指す値とSIが指す値の両方を同時にAXにロードすることになります。もちろんそんなことはできません。Idealモードでは，上の命令は次のように記述します。

```
mov  ax,[bx+si]             ; Idealモード
```

これで命令の意味は明確になります。DSが指すカレントデータセグメント内の**OFFSET BX＋SI**のメモリアドレスの内容を，AXにロードします(AXは16ビットレジスタであるため，メモリアドレスのサイズも16ビット=1ワードです。AXのかわりにALなどの8ビットレジスタを使う場合は，メモリアドレスのサイズは1バイトです)。次に同様の例を示します。

```
mov  ax,es:[bx][si]         ; MASMモード
```

この命令もIdealモードでは構文エラーとなります。この命令は「BXが指す値に対してES：セグメントオーバライドを適用し，全体をSIが指すメモリアドレスの内容に加え，その結果を無条件にAXにロードする」という意味にとれます。もちろんこれは意味を持ちません。Idealモードでは，上の命令は次のように記述します。

```
mov  ax,[es:bx+si]          ; Idealモード
```

この命令は，BXとSIレジスタを加えた値をESセグメントに対するオフセットとして(ESはデフォルトのDSをオーバライドしています)，そのメモリアドレスの内容の16ビットデータをAXにロードします。次にMASMモードでよく見かける別の例を示します。

```
mov  ax,6[bx]               ; MASMモード
```

数学者がこの例を見たなら，BXが指すアドレスの内容を6倍すると思うでしょう。あるいは，未公開の配列のインデックステクニックの一種かタイピングエラーと思うでしょうか？　実際はどれでもありません。Idealモードでは次の形式になります。

```
mov  ax,[bx+6]                ; Idealモード
```

いうまでもなく，この命令はカレントデータセグメント内のBXが指すオフセットより6バイト先のアドレスの内容をAXにロードします。MASMモードの式は，常にわかりにくいとは限りません。

```
mov  ax,es:[bp+8][si+6]     ; MASMモード
```

この命令に対しては，さまざまな解釈が可能です。BPが指すアドレスの8バイト先のアドレスから値をロードし，ESセグメントオーバライドを適用する...またはオーバライドはSIが指すアドレスの6バイト先のアドレスに適用する。あるいはBP+8の内容をロードし，[SI+6]の内容を加え，オーバライドを適用する，etc...いったいどれが正しいのかよくわかりません。このような複雑なオペランドも，Idealモードの形式を使えば一目瞭然です。Idealモードでは次のように記述します。

```
mov  ax,[es:bp+si+14]       ; Idealモード
```

この命令がESセグメント内のオフセットBP+SI+14の内容をAXにロードすることはよく理解できます。さらに別の例です。

```
mov  al,byte ptr [bx]       ; MASMモード
```

このMASMモードの命令からは，少なくともメモリアドレスの内容をバイトポインタとして指定していることはわかります。しかしIdealモードを使えば，バイトやワードはポインタ（レジスタとラベル）でのみ指し示すため，よく理解できます。

```
mov  al,[byte ptr bx]       ; Idealモード
```

この命令は，BXがバイトポインタであることを指定し，カレントデータセグメントの先頭からBXバイト先のアドレスの内容をALにロードします。さらに別の例です。

```
rep  movs byte ptr [di],[si]     ; MASMモード
```

このMASMモードの命令は，DI（またはSI？）が指すアドレスにある文字をバイトポインタにしているように思われます。しかしそのようなことはできません。Idealモードでは，次のような形式になります。

```
rep  movs [byte ptr di],[byte ptr si]     ; Idealモード
```

この命令は長いのですが，レジスタDIとSIがMOVS命令のバイトポインタになっていることがよく理解できます。

上の数例は，どれも完全なものです。MASMモードでは，オペランドの大カッコの使い方に関してわかりにくい例が他にもたくさんあります。そのような場合は，たとえ1つの命令でもIdealモードに切り換えてみてください。命令をわかりやすい形式に書き直す際は，これらの例を指標にしてください。Idealモードを使えば，より良いわかりやすいプログラムを書くための手助けとなるばかりでなく，MASMモードにおける大カッコの使い方がきわめて不明瞭であることが理解できます。

演算子

Idealモードでは，より強力・柔軟に改良された式の演算子がある一方，式の基本的な働きは変わっていません。演算子によっては，一般的な演算子の結合を容易にするためにその優先レベルが変更されているものもあります(MASMモードとIdealモードの演算子の優先レベルと詳細については，リファレンスマニュアルの第2章を参照してください)。

構造体メンバ内のピリオド

Idealモードでは，参照の対象となる構造体メンバを正確に指定する際に，構造体メンバ演算子のピリオド (.) の使い方がいっそう厳密になっています。ピリオドの左側の式は構造体ポインタで，ピリオドの右側の式は構造体のメンバ名でなければなりません。以前使用した*SomeStuff*と*OtherStuff*という構造体の例を使って，レジスタに特定の構造体メンバの値をロードする例を次に示します。

```
; 構造体の型を使って変数を宣言する
S_Stuff SomeStuff <>
O_Stuff OtherStuff <>
mov  ax,[S_Stuff.Amount]      ; ワード値をロード
mov  bl,[O_Stuff.Amount]      ; バイト値をロード
```

構造体へのポインタ

構造体のアドレス（つまりメモリ上の構造体の先頭バイトのオフセット）がストアされたレジスタを使いたい場合がしばしばあります。または構造体をアドレシングするメモリ変数を使う場合があります。これらの場合は，特定の構造体メンバを名前で参照するためには，どの構造体を参照するのかをTurbo Assemblerに知らせる必要があります。次に例を示します。

```
mov  cx,[(SomeStuff ptr bx).Amount]
```

この例は，BXが*SomeStuff*という構造体へのポインタであることと，その構造体の*Amount*フィールドの内容をCXにロードする旨をTurbo Assemblerに通知します。ピリオド演算子（.）の方が**PTR**より優先レベルが高いため，カッコ（）が必要です。カッコがないとIdealモードは*Amount*とBXを結びつけようとします。しかしレジスタはフィールド名を持つことはできないため，これはできません。構造体だけがフィールド名を持つため，構造体の参照時は構造体へのポインタをレジスタが指す構造体内のフィールドに変換しなければなりません。

SYMTYPE演算子

Idealモードのシンボルの先頭には，ピリオドを使うことはできません。またMASMモードの**.TYPE**演算子は，Idealモードでは**SYMTYPE**と呼ばれます（**リファレンスガイド**の第1章を参照してください)。この**SYMTYPE**は，未定義識別子に対して値を返さない点を除いて両モードで同じです。そのかわり，この演算子はさまざまなシンボルの型を返します。次に例を示します。

```
Abyte   DB  0
Aword   DW  0
Array   DD  20 DUP (8)
Btype   =   SYMTYPE Abyte       ; 1
Wtype   =   SYMTYPE Aword       ; 2
Atype   =   SYMTYPE Array       ; 4
```

HIGH演算子とLOW演算子

Idealモードでは，**HIGH**演算子と**LOW**演算子は2つの意味を持ちます。通常，**HIGH**は定数の上位（最上位）バイトを指定し，**LOW**は定数の下位（最下位）バイトを指定します。次に例を示します。

```
MaxVal  =   1234h
        mov ah, HIGH MaxVal     ; 12hをAHにロード
        mov al, LOW MaxVal      ; 34hをALにロード
```

Idealモードでは，メモリの参照式の上位部分と下位部分を選択するためにそれぞれ**HIGH**と**LOW**を使うことができます。次に例を示します。

```
WordVal     DW    0
DblVal      DD    0
QVal        DQ    0
      mov   bl, [BYTE LOW WordVal]
      mov   ax, [WORD HIGH DdlVal]
      mov   ax, [WORD LOW QVal]
```

最初のMOV命令は，*WordVal*にストアされた2バイト（ワード）の下位バイトをBLにロードします。次のMOV命令は，*DblVal*にストアされた4バイト（ダブルワード）の上位ワードをAXにロードします。最後のMOV命令は，*QVal*にストアされた8バイト（クオッドワード）の最下位ワードをAXにロードします。この構文は**PTR**演算子のそれと同じであることに注意してください。**BYTE**または**WORD**のキーワード，**LOW**または**HIGH**演算子，メモリの参照式の順番で並びます。

次のように**HIGH**と**LOW**を一緒に使って，複数バイト値から必要な部分だけを取り出すこともできます。

```
DVal DD    12345678h
      mov   al,[BYTE LOW WORD HIGH DVal]     ; 34hをALにロード
```

このように，キーワードの**LOW**と**HIGH**を**BYTE**や**WORD**と組み合わせることによって，変数の任意の位置からバイトやワードを取り出すことができます。この例では，*DVal*はダブルワード（4バイト）です。このような複雑な組み合わせの式は，右から左へ読むとわかりやすいでしょう。つまり上のMOV命令は，「*Dval*の**HIGH WORD**（上位ワード）の**LOW BYTE**（下位バイト）」をALにロードします。

任意指定のPTR演算子

式の中にポインタオーバライドの短縮形を使うことができます。Idealモードでは**PTR**演算子は省略できます。たとえば，

```
[BYTE PTR OverTheRainbow]
```

という式は，Idealモードでは次のような短縮形で表わすことができます。

```
[BYTE OverTheRainbow]
```

SIZE演算子

IdealモードのSIZE演算子は，データ項目が占有する実バイト数を取得します。この演算子を使えば，文字列の長さを簡単に得ることができます。次に例を示します。

```
theTitle    DB   "The Sun Also Rises"
theAuthor   DB   "Ernest Hemingway",0
titleSize   =    SIZE theTitle          ; Ideal−18, MASM−1
authorSize  =    SIZE theAuthor         ; Ideal−16, MASM−1
```

この例では，*Title*と*Author*は文字列です。MASMモードでは，**SIZE**演算子で得られる値は，名前の**LENGTH**と**TYPE**をかけたものに等しくなります。**LENGTH**は確保されているデータ項目（この例では1）と等しくなります（**LENGTH**を使うと，たとえ文字列が複数の文字を含んでいても，**DB**疑似命令による文字列は1バイトの項目とみなされます)。**DB**に対して**TYPE**演算子を使うと，やはり1が返されます。つまりMASMモードでは，*titleSize*と*authorSize*はともに1となってしまいます。これでは文字列の長さを求めたことにはなりません。

Idealモードでは，**SIZE**演算子は**DB**や**DW**などの領域確保疑似命令の直後の項目が占有するバイト数を返します。このため，*titleSize*は文字列*theTitle*の文字数に，*authorSize*は文字列*theAuthor*の文字数に等しくなります。ただし上の例の*theAuthor*は，文字列の終端を示す0バイトで終わっていることに注意してください。**SIZE**演算子が返す文字数には，このバイトは入っていません。複数の値がある場合は，**SIZE**演算子は先頭項目のみの長さを返します。次に例を示します。

```
CountDown   DB   9,8,7,6,5,4,3,2,1,"Blast off"
TwoLines    DB   "First line", 13, 10, "Second line"
CDsize      =    SIZE CountDown                      ; 1
TLsize      =    SIZE TwoLines                       ; 10
```

1行目の例の*CountDown*は9バイト値と文字列"Blast off"のあるアドレスです。この場合は，IdealモードでもMASMモードでも*CountDown*のサイズ（*CDsize*）は1（先頭要素のサイズ)になります。2行目の例の"TwoLines"ではようすが違います。この例は2つの文字列をASCIIコードの復帰（13）と改行（10）のコードで区切ってストアする典型例です。これらの2つの文字列は，プログラム内では*TwoLines*という同じラベルに属します。この例では，**SIZE**演算子は先頭項目つまり"*First Line*"の長さを返します。したがって*TLsize*（*TwoLines*の長さ）の値は，Idealモードでは10（先頭文字列の文字数）になり，MASMモードでは1（先頭の**DB**要素のサイズ=1バイト=1文字）になります。

疑似命令

Idealモードでは疑似命令は理想的に機能します。疑似命令の名前は，ほとんどの場合MASMモードでの名前と同じですが，両モードで名前が異なるまぎらわしい疑似命令もいくつかあります。次にそれらの疑似命令について説明します。

リスト制御疑似命令

Idealモードではシンボル名の先頭にピリオド（.）を使うことができないため，MASMモードのリスト制御疑似命令の先頭文字はパーセント記号（%）となります。また，処理の内容を正確に表わすために，名前が変わる疑似命令もあります。次表に両モードのリスト制御疑似命令を示します。

MASMモード	**Idealモード**
.CREF	%CREF
.LALL	%MACS
.LFCOND	%CONDS
.LIST	%LIST
.SFCOND	%NOCONDS
.XALL	%NOMACS
.XCREF	%NOCREF
.XLIST	%NOLIST

Idealモードでは，すべてのリスト制御疑似命令はパーセント記号（%）で始まるため，MASMモードの**%OUT**疑似命令(メッセージ表示用疑似命令)はIdealモードでは**DISPLAY**に変更されています。次に例を示します。

```
DISPLAY "Starting to Assemble I/O Driver"
```

ピリオド（.）で始まる疑似命令

ピリオド(.)で始まる他のMASMモード疑似命令も，わかりやすいように名前が変更されています。たとえばプロセッサ制御疑似命令では，**.286**(これは疑似命令というよりも数

値に見えます)は**P286N**という具合に変更されます。また**.ERR*xxx***の形式を持つ強制エラー疑似命令は，**ERRIF*xxx***という形式になります。この他の疑似命令では，先頭のピリオドがなくなっているものもあります。

ピリオドで始まるMASM疑似命令とIdealモードの対応疑似命令を次表に示します。

MASMモード	Idealモード	MASMモード	Idealモード
.186	P186	.ERR2	ERRIF2
.286	P286N	.ERRB	ERRIFB
.286C	P286N	.ERRDEF	ERRIFDEF
.286P	P286	.ERRDIF	ERRIFDIF
.287	P287	.ERRDIFI	ERRIFDIFI
.386	P386N	.ERRE	ERRIFE
.386C	P386N	.ERRIDN	ERRIFIDN
.386P	P386	.ERRIDNI	ERRIFIDNI
.387	P387	.ERRNB	ERRIFNB
.8086	P8086	.ERRNDEF	ERRIFNDEF
.8087	P8087	.ERRNZ	ERRIF
.CODE	CODESEG	.FARDATA	FARDATA
.CONST	CONST	.FARDATA?	UFARDATA
.DATA	DATASEG	.MODEL	MODEL
.DATA?	UDATASEG	.RADIX	RADIX
.ERR	ERR	.STACK	STACK
.ERR1	ERRIF1		

順序が反転した疑似命令とシンボル名

Idealモードのパーシングの順序は，MASMモードのそれよりも単純です。先頭のトークンがキーワードの場合は，処理は疑似命令が実行するとみなされます。キーワードでない場合は，2番目のトークンによって処理が決まります。

このような変更があるため，処理によっては疑似命令とシンボル名の順番が反転する場合があります。次表にそれらを示します。

MASMモード	Idealモード
name ENDP	**ENDP** [name]
name ENDS	**ENDS** [name]
name GROUP *segs*	**GROUP** name segs
name LABEL *type*	**LABEL** name type
name MACRO *args*	**MACRO** name args
name PROC *type*	**PROC** name type
name RECORD *args*	**RECORD** name args
name SEGMENT *args*	**SEGMENT** name args
name STRUC	**STRUC** *name*
name UNION	**UNION** *name*

Idealモードでは，**ENDS**と**ENDP**でシンボル名（セグメント名やプロシージャ名）が省略できることに注意してください。シンボル名を指定する場合は，直前の**SEGMENT**や**PROC**疑似命令で指定したものと同じ綴りのものを使います。プログラマによっては，常にシンボル名を記述しプログラムを読みやすくすることに努めている人もいます。特にプロシージャやセグメントをネスト構造にする場合は，常にシンボル名を記述した方が良いでしょう。必要がなければ省略することもできます。

MASMモードとIdealモードでまったく変更のない疑似命令もあります。たとえば次の疑似命令は，シンボルを言語構文の一部として定義するためどちらのモードでも同じです。

=	**DD**	**DQ**
:	**DF**	**DT**
DB	**DP**	**DW**
		EQU

引用符で囲んだ文字列を伴う疑似命令

INCLUDE疑似命令は，Idealモードで使う場合はファイル名を引用符で囲みます。

```
INCLUDE "MYDEFS.INC"
```

MASMモードで使う場合はファイル名を引用符で囲む必要はありません。

```
INCLUDE MYDEFS.INC
```

%TITLEや**%SUBTTL**では，タイトル文字列を引用符で囲む必要があります。

```
%TITLE   "Macro Definitions"              ; コメントは無視される
%SUBTTL  "Block Structuring Macros"       ; コメントは無視される
```

これらの2例からわかるとおり，Idealモードではタイトルやサブタイトルが引用符で囲まれているため，タイトル行の最後にコメントを付け加えることができます。この場合，コメントはリストファイルには入りません。MASMモードでは，**.TITLE**や**.SUBTTL**疑似命令の後にある文字列はすべてタイトル文字列とみなされます。

セグメントとグループ

Turbo AssemblerがIdealモードでセグメントやグループを扱う方法によって，プログラム実行のようすが違ってきます。普通のプログラマなら，セグメントとグループの相互作用に起因するバグを扱うと思うとぞっとすることでしょう。

このバグを見つけることが困難なのは，MASMおよびTurbo AssemblerのMASMモードでのグループ内データ・コードの参照に関する予想の仕方に原因があります。幸運にもIdealモードでは，MASMのセグメントとグループの疑似命令に起因するめんどうな問題が軽減されています。以下にこれについて説明します。

グループに属するセグメント内データのアクセス

Idealモードでは，グループに属するセグメント内の項目は，セグメントではなく基本的にはそのグループのメンバとみなされます。Turbo Assemblerにデータ項目をセグメントのメンバとして認識させるためには，明示的なセグメントオーバライドを使う必要があります。

MASMモードでは，次のように対処しています。つまり，シンボルをグループではなくセグメントの一部とみなします。特にMASMモードでは，シンボルは**OFFSET**とともに使われる場合はセグメントの一部として扱われ，データ領域内のポインタとして使われる場合はグループの一部として扱われます。**OFFSET**を使わずに直接データをアクセスする場合，MASMはグループではなくセグメントに基づいて相対参照するため，これは混乱の元になります。 MASMのあいまいなアドレシングを使うと，たちまちトラブルに巻き込まれることを次に例を使って示します。3つのデータセグメントを宣言する次の不完全なMASMプログラムを考えてみましょう。

```
dseg1   SEGMENT para public 'data'
v1      db      0
dseg1   ENDS

dseg2   SEGMENT para public 'data'
v2      db      0
dseg2   ENDS

dseg3   SEGMENT para public 'data'
v3      db      0
dseg3   ENDS

DGROUP  GROUP   dseg1,dseg2,dseg3
cseg    SEGMENT para public 'code'

        ASSUME  cs:cseg,ds:DGROUP

start:
        mov     ax,OFFSET v1
        mov     bx,OFFSET v2
        mov     cx,OFFSET v3
cseg    ENDS
        END     start
```

この例では，3つのセグメント*dseg1, dseg2, dseg3*が**DGROUP**という1つの名前でグループ化されています。その結果，各セグメント内のすべての変数は一緒にメモリにストアされます。このプログラムは，各セグメント内で*v1, v2, v3*というラベルのバイト変数を宣言しています。

このMASMプログラムは，これらの3つの変数のオフセットアドレスをそれぞれAX, BX, CXレジスタにロードします。先に**ASSUME**疑似命令が使われ，3つのデータセグメントがグループ化されているため，MASMは変数がストアされたグループ内の相対アドレスに基づいて変数のオフセットを計算するように思われます。

しかし実際にMASMは，予想に反して*seg1, seg2, seg3*の各セグメント内の相対アドレスに基づいて変数のオフセットを計算します。この例では，これら3つのセグメントを連結してメモリ上の1つのデータセグメントとして扱い，DSでアドレシングしています。複数セグメントがこのようにメモリ上で1つのセグメントとしてまとめられている場合は，各セグメントに対する相対アドレスを変数のオフセットとしても無意味です。このような変数をアドレシングする唯一の方法は，グループ内の相対アドレスをオフセットとすることです。

MASMモードにおけるこの問題を解決するには，**OFFSET**演算子を使ってグループ名を指定することが必要です。次に例を示します。

```
mov    ax,OFFSET DGROUP:v1
mov    bx,OFFSET DGROUP:v2
mov    cx,OFFSET DGROUP:v3
```

この命令は正しくアセンブルされ，**DGROUP**（複数セグメントをまとめたグループ）の

相対アドレスに基づいた*v1, v2, v3*のオフセットをレジスタにロードします。しかし識別名**DGROUP**の指定を忘れることはよくあります。このようなミスをおかした場合は，オフセット値はメモリ上の変数のロケーションを正しく示しません。しかもMASMはその旨を通知しません。Idealモードでは，このようなトラブルは起こりません。Idealモードのプログラムは次のようになります。

```
         IDEAL
SEGMENT dseg1   para public 'data'
v1       db     0
ENDS

SEGMENT dseg2   para public 'data'
v2       db     0
ENDS

SEGMENT dseg3   para public 'data'
v3       db     0
ENDS

GROUP   DGROUP   dseg1,dseg2,dseg3
SEGMENT cseg     para public 'code'

         ASSUME  cs:cseg,ds:DGROUP

start:
         mov    ax,OFFSET v1
         mov    bx,OFFSET v2
         mov    cx,OFFSET v3
ENDS
         END    start
```

*v1, v2, v3*のオフセットは，グループ（変数が属するセグメントどうしをまとめたもの）内の相対アドレスとして正しく計算されます。Idealモードでは，グループ化されたセグメント内の変数を参照するために識別名**DGROUP**は必要ありません。しかし，MASMモードではこの識別名が必要となります。さらに悪いことに，たとえプログラマがグループ名を指定し忘れても何の警告メッセージも出力しません。

near型またはfar型のコードラベルの定義

near型とfar型の**LABEL**シンボルまたは**PROC**シンボルを定義する場合は，シンボルの参照はシンボルのセグメントを含むグループ内のオフセットに基づいて行なわれます。シンボルのセグメントがグループの構成セグメントでない場合は，シンボルはセグメント内のオフセットに基づいて参照されます。したがってIdealモードでは，nearまたはfar型のシンボルを定義するために**ASSUME CS**疑似命令を使ってセグメントを想定する必要はあり

ません。次にMASMモードにおける例を示します。

```
CODE SEGMENT
     ASSUME   CS:CODE
XYZ  PROC FAR
     .
     .                         ; MASMモードのプロシージャコード
     .
XYZ  ENDP
CODE ENDS
```

このMASMモードのプログラムは，Idealモードでは次のようになります。

```
SEGMENT CODE
PROC XYZ FAR
     .
     .                         ; Idealモードのプロシージャコード
     .
ENDP
ENDS
```

この変更によってMASMモードに新しい機能がもたらされるわけではありませんが，通常Idealモードでは自動的に解決されることをTurbo Assemblerに指示する必要はなくなります。

外部シンボル，パブリックシンボル，グローバルシンボル

EXTRNや**GLOBAL**疑似命令を使って型（**BYTE, WORD**など）を指定する必要のある場合は，いつでも構造体名を使うことができます。次に例を示します。

```
STRUC    MoreStuff
HisStuff  DB   0
HerStuff  DW   0
ItsStuff  DB   0
ENDS
EXTRN   SNAME:MoreStuff
```

先ほど説明したピリオド（.）演算子とこの機能を組み合わせることによって，ソースモジュールの外部にある構造体メンバを参照することができます。これによって両モジュール内でメンバを宣言したことになります。**SIZE**演算子は外部データ構造のサイズも通知します。Idealモードでは，いったん指定したパブリックシンボルは**PUBLIC**を指定した場所で発生します。これは変数を再定義する際に役に立ちます。MASMモードではすべてのパブリックシンボルはプログラムの終わりに付加されるため，パブリックシンボルの再定義の方法に制限を与えます。次に例を示します。

```
Perfect  =  8
    PUBLIC   Perfect      ; Perfectをパブリック宣言する
Perfect  =  10            ; Perfectの値を再定義する
```

Idealモードでは，モジュールはPUBLIC宣言の後で*Perfect*を再定義していても*Perfect*の値は8です。MASMモードではPUBLICシンボルがモジュールの最後に付けられるため，このシンボルをEXTRN宣言によってインポートする他のモジュールは*Perfect*の値として10を受け入れます。

その他の相違点

このセクションでは，MASMモードとIdealモードのその他の相違点について説明します。

余計な領域を確保しない

IdealモードのTurbo Assemblerでは，ページまたはパラグラフでアラインメントされたプライベートセグメントに対してはセグメントに関係する領域を確保しません。リンカはこのような領域を必要としないため，Idealモードでプログラムをアセンブルするとモードよりオブジェクトファイルのサイズが小さくなり，これらのオブジェクトファイルはMASMモードで作られるオブジェクトファイルよりも高速でリンクされます。次の例は，MASMモードではいかに余計な領域が確保されるかを示すものです。

```
SEGMENT DATA PRIVATE PARA
VAR1 DB   0
VAR2 DW   0
ENDS
SEGMENT CODE
     ASSUME ds:DATA
     mov  ax,VAR2            ; 領域必要なし
ENDS
```

プログラムの記述自体は変わりありません。コメントはほんの参考にすぎません。

BOUND命令のオペランド

BOUND命令では，DWORDではなくWORDのオペランドの使用を前提としています。したがって範囲の下限値と上限値を2つのワード定数で定義することができ，明示的にDWORD PTRを使ってオペランドをDWORDに変換する必要はありません。MASMモードでは次にように記述しなければなりません。

```
BOUNDS  DW  1,4                     ; 下限値と上限値
        BOUND   DWORD PTR BOUNDS    ; MASMモードでは必要
```

Idealモードでは次のようになります。

```
BOUNDS  DW  1,4                     ; 下限値と上限値
        BOUND     [BOUNDS]          ; Idealモードでは有効
```

マクロ内のコメント

Idealモードでは，マクロ内のコメントは文字列として扱われます。マクロのコメント内でダミーパラメータを表わすには，パラメータの前にアンパサンド(&)を付けなくてはなりません。

```
MACRO DOUBLE ARG
     SHL  arg,1        ; &ARGを2倍する
ENDM
```

IdealモードでDOUBLE BXという形でこのマクロを使う場合は，リストファイルは次のようになります。

```
SHL bx,1               ; BXを2倍する
```

一方マクロを次のように定義すると，

```
MACRO DOUBLE arg
     SHL  arg,1        ; ARGを2倍する
ENDM
```

リストファイル内でARGは置き換えられません。

```
SHL bx,1               ; ARGを2倍する
```

ローカルシンボル

Turbo Assemblerのローカルシンボル機能は，Idealモードに切り換えた時点で自動的に有効になります。これは**LOCALS**疑似命令を指定することと同じです。

MASMモードとIdealモードのプログラミングの比較

この章の最後に，MASMモードとIdealモードの違いを示すために両モードにおけるサンプルプログラムを載せます。これらのプログラムをよく読んだ後，さらに最後の番号付きの説明を読めばIdealモード構文の利点を理解することができます。

これらのプログラムは，プログラミングスタイルの見本を示すための例ではありません。この章で説明したIdealモードの概念とIdealモードの機能一般とMASMとの違いを示します。

サンプルプログラムは，コンソールから1行を読み込み，テキストを大文字に変換し，結果を画面に表示してDOSに戻ります。MASMモードとIdealモードのプログラムコードの違いを示すために，Idealモードのプログラムにはコメント（セミコロンと番号）を付けてあります。たとえばコメントに#4とあれば，プログラム例の後の「MASMモードとIdealモードの分析」の節の対応番号の説明を読んでください。またIdealモードの特徴を目立たせるために，プログラムからはほとんどのコメントを取り除いています。MASMモードのサンプルプログラムを読んでプログラムの機能を把握し，Idealモードのサンプルプログラムを読んで両モードの違いを比較してください。

MASMモードのサンプルプログラム

```
; ファイル <masexmpl.asm>
; MASMモードのサンプルプログラム（行を大文字に変換する）
     TITLE  Example MASM Program  ; このコメントはタイトルに含まれる!
     .286

bufsize  =  128                   ; 入力/出力バッファのサイズ

dosint MACRO intnum
     mov  ah,intnum               ; ファンクション番号をAHに割り当てる
     int  21h                     ; DOSファンクション&INTNUMをコール
ENDM

stk SEGMENT STACK
     db   100h DUP (?)            ; スタック領域を確保
stk ENDS

data SEGMENT WORD
inbuf     db   bufsize DUP (?)    ; 入力バッファ
outbuf    db   bufsize DUP (?)    ; 出力バッファ
data ENDS
```

```
DGROUP GROUP stk,data               ; グループのスタックセグメントとデータセグメント

code SEGMENT WORD
     ASSUME   cs:code               ; CSはコードセグメントとする
start:
     mov  ax,DGROUP                 ; DGROUPのアドレスを割り当てる
     mov  ds,ax                     ; DSにセグメントをロード
     ASSUME   ds:DGROUP             ; デフォルトデータセグメントはDS
     mov  dx,OFFSET DGROUP:inbuf    ; DXにinbufのオフセットをロード
     xor  bx,bx                     ; 標準入力
     call readline                  ; 1行読み込み
     mov  bx,ax                     ; BXに長さをロード
     mov  inbuf[bx],0               ; 終端にヌルをストア
     push ax                        ; AXをスタックへ退避
     call mungline                  ; 行を大文字に変換
     pop  cx                        ; カウントを復帰
     mov  dx,OFFSET DGROUP:outbuf   ; DXにoutbufのオフセットをロード
     mov  bx,1                      ; 標準出力
     dosint     40h                 ; ファイル書き込みファンクション
     dosint     4ch                 ; DOSへ戻る

; 行を読み込む....呼び出し:dx=バッファ / リターン値: AX=カウント
readline PROC near
     mov  cx,bufsize                ; バッファサイズを指定
     dosint     3fh                 ; ファイル読み込みファンクション
     and  ax,ax                     ; カウントにゼロフラグをセット
     ret                            ; リターン
readline ENDP

; 行を大文字に変換する
mungline PROC NEAR
     mov  si,OFFSET DGROUP:inbuf    ; SIにinbufのアドレスをロード
     mov  di,0                      ; DIを初期化
@@uloop:
     cmp  BYTE PTR[si],0            ; テキスト終了?
     je   @@done                    ; YES...@@doneへジャンプ
     mov  al,[si]                   ; NO....次の文字を取得
     and  al,not 'a'-'A'            ; 大文字に変換
     mov  outbuf[di],al             ; 出力バッファへストア
     inc  si                        ; lodsb,stosbを使う方がよい
     inc  di                        ; ...これは例です
     jmp  @@uloop                   ; テキスト変換を繰り返す
@@done:   ret
mungline ENDP                       ; プロシージャの終わり
code ENDS                           ; コードセグメントの終わり
     END start                      ; テキストの終り/DOSのエントリポイント指定
```

Idealモードのサンプルプログラム

```
; ファイル <masexmpl.asm>
; MASMモードのサンプルプログラム (行を大文字に変換する)
     IDEAL                                  ; #1
     %TITLE "Example Ideal-Mode Program"    ; #2
     P286N                                  ; #3

Bufsize  =  128

MACRO dosint intnum                         ; #4
     mov  ah,intnum
     int  21h
ENDM

SEGMENT stk STACK                           ; #5
     db   100h DUP (?)
ENDS                                        ; #6

SEGMENT DATA WORD                           ; #7
inbuf      db   BufSize DUP (?)
outbuf     db   BufSize DUP (?)
ENDS DATA                                   ; #8

GROUP DGROUP stk,DATA                       ; #9

SEGMENT CODE WORD                           ; #10
     ASSUME   cs:code
start:
     mov  ax,DGROUP
     mov  ds,ax
     ASSUME   ds:DGROUP
     mov  dx,OFFSET inbuf                   ; #11
     xor  bx,bx
     call readline
     mov  bx,ax
     mov  [inbuf + bx],0                    ; #12
     push ax
     call mungline
     pop  cx
     mov  dx,OFFSET outbuf                  ; #13
     mov  bx,1
     dosint    40h
     dosint    4ch

; 行を読み込む....呼び出し: dx = バッファ / リターン値: AX=カウント
PROC readline near                          ; #14
     mov  cx,Bufsize
     dosint    3fh
     and  ax,ax
     ret
ENDP                                        ; #15
```

```
; 行を大文字に変換
PROC mungline NEAR                              ; #16
     mov  si,OFFSET inbuf                       ; #17
     mov  di,0
@@uloop:
     cmp  [BYTE si],0                           ; #18
     je   @@done
     mov  al,[si]
     and  al,not 'a'-'A'
     mov  [outbuf + di],al                      ; #19
     inc  si
     inc  di
     jmp  @@uloop
@@done:   ret
ENDP mungline                                   ; #20
ENDS                                            ; #21
     END start
```

MASMモードとIdealモードの分析

上記MASMモードとIdealモードのサンプルプログラムにおける構文, 疑似命令, 演算子の違いを詳しくまとめておきます。番号はIdealモードのサンプルプログラムのコメントに対応しています。これらの行をMASMモードのサンプルプログラムと比較してください。

1. **IDEAL**疑似命令を使ってIdealモードにします。Turbo Assemblerでは，特に指定のない限り常にMASMモードで開始します。**MASM**疑似命令は, IdealモードからMASMモードに戻る場合にのみ使います。
2. **%TITLE**の先頭にはパーセント記号（%）があるため，この疑似命令がリストファイルに影響を与える疑似命令であることがわかります（リストファイルを作るには，アセンブル時に/**L**コマンドラインオプションを使ってリストファイル名を指定します)。Idealモードでは**TITLE**（%なし）のかわりに**%TITLE**を使い，タイトル文字列は引用符（""）で囲みます。これによってタイトル行にコメントを付けることができます(MASMモードでは，**TITLE**疑似命令の後の文字列はすべてタイトルとみなされます)。
3. MASMモードの**.286**疑似命令は，Idealモードでは**P286N**となっています。Idealモードではシンボルの先頭にピリオド(.)を付けられないため，ピリオドで始まるMASMモードの疑似命令はすべて変更されています。このプログラムでは，文は両モードの違いを示すこと以外には特に役に立っていません。このプログラムでは80286の命令は使っていません。

4. Idealモードの**MACRO**疑似命令では，マクロは疑似命令の後に書きます(MASMモードでは逆です)。
5. Idealモードの**SEGMENT**疑似命令では，セグメント名は疑似命令の後に書きます。
6. Idealモードでは，セグメントを終了する**ENDS**疑似命令の後にMASMモードのようにセグメント名を指定する必要はありません（好みによってセグメント名を指定しても問題ありません)。
7. 5と同様。**SEGMENT**キーワードはセグメント名より先に書きます。
8. **ENDS**疑似命令にセグメント名を指定する場合は，疑似命令の後に書きます(MASMモードでは疑似命令の前です)。もちろん省略することもできます。
9. Idealモードでは，**GROUP**疑似命令がデータセグメントグループ（**DGROUP**）に先行します。この後にグループ化したいデータセグメントの一覧を記述します。MASMモードでは**GROUP**と名前の順番は逆です。
10. 5と同様。**SEGMENT**キーワードはセグメント名より先に書きます。
11. **OFFSET**演算子と一緒にグループ識別名を使う必要はありません。Idealモードでは，**INBUF**は**DGROUP**の先頭からのオフセットとなります。なぜなら**INBUF**はこのグループ名のもとにまとめられたセグメントのうちのいずれかのセグメント内にあるからです。MASMモードでグループ化セグメント内の変数オフセットを正しく示すには，必ず***DGROUP:inbuf***を指定しなければなりません。
12. [INBUF+BX]オペランドは，MASMモードとIdealモードで有効です。しかしMASMモードの**INBUF**[BX]はIdealモードでは無効です。Idealモードでは，すべてのメモリ参照オペランドは大カッコで囲まなくてはなりません。
13. 11と同様。グループ化セグメント内の変数を参照するためにグループ名を指定する必要はありません。MASMモードでは，**OUTBUF**の正しいオフセットを求めるためには**DGROUP:outbuf**を記述しなければなりません。このプログラムで**DGROUP**識別名を忘れると，データをスタックにストアしてしまいます。にもかかわらず，MASMはそのような重大な間違いに対して警告メッセージを出力しません。
14. **PROC**疑似命令では，プロシージャ名は疑似命令の後に書きます(MASMモードでは疑似命令の前です)。
15. Ideal モードで**ENDP**を使ってプロシージャを終了する場合は，MASMモードのようにプロシージャ名を指定する必要はありません。
16. 14と同様。**PROC**疑似命令はプロシージャ名に先行します。
17. 11と同様。MASMモードのように**DGROUP:inbuf**を記述する必要はありません。
18. Idealモードでは，式のサイズを設定すれば**PTR**演算子は省略できます。MASMモードの**BYTE PTR ABC**は，Idealモードの**BYTE ABC**と同じです。

19. 12と同様。Idealモードでは，メモリの定数を参照する場合は常に大カッコ内にメモリの参照式を入れます。

20. **ENDP**疑似命令の後に必要に応じてプロシージャ名を付けます(MASMモードでは疑似命令の前に付けます)。

21. 6.と同様。**ENDS**疑似命令にはセグメント名を付ける必要はありません。好みに応じて付けることもできます。

参考文献

Crawford, John H., and Patrick P. Gelsinger. *Programming the 80386*. Alameda: Sybex, Inc., 1987（岩谷宏訳「**80386プログラミング**」工学社）

Duncan, Ray. *Advanced MS-DOS*. Redmond: Microsoft Press, 1986.

Duncan, Ray. *The MS-DOS Encyclopedia*. Redmond: Microsoft Press, 1988.
（「MS-DOSエンサイクロペディア　Volume 1,2」アスキー）

Lafore, Robert. *Assembly Language Primer for the IBM PC & XT*. New York: The Waite Group, 1984

Murray, William H., and Chris Pappas. *80386/80286 Assembly Language Programming*. Berkeley: Osborne/McGraw-Hill, 1986

Norton, Peter, and John Socha. *Peter Norton's Assembly Language Book for the IBM PC*. New York: Brady Communications, 1986

Rector, Russell, and George Alexy. *The 8086 Book*. Berkeley: Osborne/McGraw-Hill, 1980（吉川敏則訳「**ザ8086ブック**」秋葉出版）

Sargent III, Murray, and Richard L. Shoemaker. *The IBM PC from the Inside Out*. Reading: Addison-Wesley, 1986.

Skinner, Thomas P. *An Introduction to Assembly Language Programming for the 8086 Family*. New York: John Wiley & Sons, Inc., 1985

Turley, James L. *Advanced 80386 Programming Techniques*. Berkeley:Osborne/McGraw-Hill, 1988.

Wilton, Richard. *Programmer's Guide to PC and PS/2 Video Systems*. Redmond: Microsoft Press, 1987.

「**iAPX86ファミリ・ユーザーズマニュアル**」インテルジャパン
（このほかに，各プロセッサチップのユーザーズマニュアルあるいはプログラマーズマニュアルはIntel Corporationより，また邦訳はインテルジャパン社より発行されています）。

アスキー出版局テクライト編「**PC-9800シリーズ テクニカルデータブック**」
アスキー出版局

アスキー出版局テクライト編「**PC-9800シリーズ テクニカルデータブック 増補改訂2版**」
アスキー出版局

索　引

.(ピリオド)
Idealモード 564,570
疑似命令の～ 574
.(ピリオド)演算子 468
;; 演算子(マクロ内) 463
?(疑問符) 206
初期値不設定記号 146
ラベル内の～ 97
??Date変数 206
??Filename変数 206
??Time変数 206
! 演算子(マクロ内) 463
¥(円記号) 229
=疑似命令 70,196
$(ドル記号)
ラベル内の～ 97
%(パーセント記号)
Idealモードの疑似命令と～ 574
%演算子(マクロ内) 463
#pragma指令(Turbo C) 301
&(アンパサンド)
パラメータと～ 458
&演算子(マクロ内) 463
@(単価記号) 91
ローカルラベルと～ 438
@CodeSizeシンボル 126
@cursegシンボル 126
@dataシンボル 121
@fardataシンボル 126
@fardata?シンボル 126
@FileNameシンボル 126
-1オプション(Turbo C) 304
2進法 135
2の補数 155
2パスアセンブラ
～との互換性 256
8進法 135
10進法 135
10バイト整数 135
16進法 43,135
16ビット整数 134
32ビット演算 163
32ビット整数 134
32ビットポインタ(Turbo Prolog) 415
64ビット整数 135
.186疑似命令 399,507
.286疑似命令 516
.286C疑似命令 516
.286P疑似命令 516
.287疑似命令 399
.386疑似命令 399,517
.386C疑似命令 517
.386P疑似命令 517
.387疑似命令 558
.8086疑似命令 506
8086の算術演算 162
8086プロセッサ
I/O 29
アーキテクチャ 26
計算能力 162
命令セット 49
メモリ 27
レジスタ 31
8087コプロセッサ 61,71
Turbo Cと～ 344
8088プロセッサ 23,26,505,506
80186プロセッサ 507-514
BOUND命令 509
ENTER命令 508

INS命令 510
LEAVE命令 508
OUTS命令 510
POPA命令 507
PUSHA命令 507
Turbo Basicと～ 399
Turbo Cと～ 304
拡張命令 511
シフトとローテート 512
即値の乗算 513
即値のプッシュ 512
プロテクトモード 82
80188プロセッサ 507
80286プロセッサ 515
Turbo Basicと～ 399
Turbo Cと～ 305
保護モード 82,515
メモリ管理 515
モード 515
レジスタ 515
80287コプロセッサ 557
保護モード 557
80386プロセッサ 517-556
6バイト変数 524
16ビットセグメントと
32ビットセグメントの混在 550
32ビット命令 545
48ビット型 523
DF疑似命令 524
LOOP命令 546
MOV命令 544
値の変換 539
新しい命令 536
アドレシングモード 531
条件テスト 542
乗算 549
ストリング命令 547
セグメント疑似命令 522
特権命令 517
ビットのスキャン 537
ビットのテスト 536
複数ワードにまたがるシフト 540
ポインタの設定 543
レコードサイズ 473
レジスタ 526
80387コプロセッサ 558
新しい命令 558

【A】

/Aオプション 68,84
ABS型 224
ADC命令 163
ADD命令 162
キャリフラグへの影響 280
AHレジスタ 34
除算 168
ALレジスタ 34
I/O 160
値を返す 191
乗算 166
除算 168
.ALPHA疑似命令 488
ALPHA疑似命令 68,84
AND命令 169
TEST命令と～ 477
レコードと～ 475
ANSI.SYS 61
ARG疑似命令
Turbo Cと～ 341
asm(キーワード) 295

.ASMファイル 1,65
ASSUME疑似命令 127,219,491
　CSレジスタと～ 491
　グループ名と～ 489
AT結合型 484
AXレジスタ 34
　I/O 160
　値を返す 191
　乗算 167
　除算 167,168

【B】

/Bオプション 67
-bオプション(Turbo C) 301
BASIC → Turbo Basic
BCD 135
BHレジスタ 35
%BIN疑似命令 245
.BINファイル
　スタックと～ 263
BIOS 59
　DSセグメント 123
　カラー 60
　画面モード 60
BLレジスタ 35
BOUND命令
　80186 509
　Idealモード 582
BPレジスタ 39
　メモリオペランドと～ 432
　メモリポインタ 113,115
BSF命令 538
BSR命令 538
BT命令 536
BTC命令 536
BTR命令 536
BTS命令 536
BXレジスタ 35,48
　除算 168
　メモリポインタ 112
BYTE型 147
BYTE PTR演算子 153

【C】

C → Turbo C
/Cオプション 69
CALL命令
　サブルーチンと～ 186
　ラベルと～ 108
CALLルーチン(Turbo Basic) 398
CBW命令 156
CDQ命令 539
CGA 54
CHレジスタ 37
CLレジスタ 37
　シフト 173
　ローテート 175
CLD命令 208,272
CMP命令
　SCAS命令と～ 214,217
CMPS命令 217,273
　ESレジスタと～ 432
　～の繰り返し 273
CMPSB命令 217
CMPSW命令 217
.CODE疑似命令 120
.COMファイル
　Turbo Basicと～ 399
　スタックと～ 263
COMMON結合型 485

%CONDS疑似命令 242
.CONST疑似命令 126,497
.CREF疑似命令 241
%CREF疑似命令 240,241
CSEG 358
CSオーバライド 82
CSレジスタ 48
ASSUME疑似命令と～ 128,491
サブルーチンと～ 188
%CTLS疑似命令 243
CWD命令 157
CWDE命令 539
CXレジスタ 35
CXレジスタのゼロ
LOOP命令と～ 546
繰り返しストリング命令と～ 271,273
ストリング命令と～ 271,273
ループ処理と～ 183

【D】

/Dオプション 70,251
.DATA?疑似命令 126,497
.DATA疑似命令 120
ESレジスタと～ 122
DB疑似命令 142
DD疑似命令 139,142
DEC命令 165
キャリフラグへの影響 281
%DEPTH疑似命令 245
DF疑似命令 142
80386 524
DGROUP 490
DHレジスタ 37
DIレジスタ 38,48
ストリング命令 213
メモリポインタ 113
DISPLAY疑似命令 247
DIV命令 167
DLレジスタ 37
DOS 56
PSP 358
キーの読み込み 57
～コール 56
～に戻る 258
プログラムの終了 59
文字の表示 58
ワイルドカード 65
DOSSEG疑似命令 124,487
Turbo Cと～ 323
DP疑似命令 142
DQ疑似命令 139,142
DSレジスタ 48
ASSUME疑似命令と～ 129
BIOSと～ 123
.DATA疑似命令と～ 121
ストリング命令と～ 273
メモリオペランドと～ 432
DSEG 358
DT疑似命令 139,142
DUP演算子 143
REPTと～ 451
構造体と～ 468
DW疑似命令 142
DWORD型 147
DXレジスタ 37
I/O 160
乗算 167
除算 168

【E】

/Eオプション 71
EAXレジスタ 527
EBPレジスタ 527
EBXレジスタ 527
ECXレジスタ 527
EDIレジスタ 527
EDXレジスタ 527
ELSE疑似命令 254
ELSEIF疑似命令 254
EMUL疑似命令 71
END疑似命令 95,101
　開始アドレスと〜 102
ENDIF疑似命令 248
ENDM疑似命令 450
ENDP疑似命令
　Idealモード 576
　サブルーチンと〜 188
ENDS疑似命令 127,480,497
　Idealモード 576
ENTER命令(80186) 508
EQU疑似命令 200
　IdealモードとMASMモード 560,566
　不等号カッコと〜 204
.ERR疑似命令 255
.ERR1疑似命令 255
.ERR2疑似命令 255
.ERRB疑似命令 257
.ERRDEF疑似命令 256
.ERRE疑似命令 256
.ERRNB疑似命令 257
.ERRNDEF疑似命令 256
.ERRNZ疑似命令 256
ESレジスタ 47
　ASSUME疑似命令と〜 129
　.DATA疑似命令と〜 123
　LES命令と〜 543
　ストリング命令と〜 273,432
ESIレジスタ 527
ESPレジスタ 527
.EXEファイル 1,11
EXITM疑似命令 461
EXTRN疑似命令 223
　IdealモードとMASMモード 581
　Turbo Cと〜 333
　Turbo Pascalと〜 365

【F】

FAR型 148
　FAR PTR演算子と〜 154
farサブルーチン 189
farデータ 124,126
　Idealモード 580
.FARDATA?疑似命令 126,499
.FARDATA疑似命令 126,498
FAR PTR演算子 154
　前方ジャンプと〜 447
FCOS命令 558
FPREM1命令 558
FSレジスタ 529
　LES命令と〜 543
FSETPM命令 558
FSIN命令 558
FSINCOS命令 558
FUCOM命令 558
FUCOMP命令 558
FUCOMPP命令 558
FWORD型 147

80386 524

【G】

GLOBAL疑似命令 226
Idealモードとの MASMモード 580
GREP.COM 6
GROUP疑似命令 489
IdealモードとMASMモード 579
GSレジスタ 529
LES命令と～ 543

【H】

/Hオプション 72
HIGH演算子
Idealモード 571

【I】

/Iオプション 73,229
I/O 160
8086 29
80186 510
ALレジスタ 160
AXレジスタ 160
DXレジスタ 37,160
IBM PC 54
キーボード 57
処理 22
フォーマット化～ 58
iAPx86プロセッサファミリ 26,505
IBM PC
I/O 54
システムソフトウェア 55
特長 54
IBM XT→IBM PC
IDEAL疑似命令 562
Idealモード 1,559-589
BOUND命令 582
near/farデータ 579
イコール定義と～ 566
演算子 570
オペランド 568
外部シンボル 565
疑似命令 574
グループオーバライド 289
構造体/共用体 564,580
式 568
セグメントグループ 577
セグメントの付加デバッグ情報 581
速度 561
トークン 564
浮動小数点数 566
マクロ内のコメント 582
リスト制御 574
ローカルシンボル 583
ローカルラベル 439
IDIV命令 169
IF疑似命令 249
IF1疑似命令 254
IF2疑似命令 254
IFB疑似命令 252
IFDIF疑似命令 253
IFE疑似命令 249
IFIDN疑似命令 253
IFNB疑似命令 252
IMUL命令 167,280
80186 513
80386 549
問題点 279-280
IN命令 37,160
INC命令 165

キャリフラグへの影響 280
%INCL疑似命令 242
INCLUDE疑似命令 73,228
Idealモード 576
INS命令(80186) 510
INT命令
キーボード入力と～ 57
文字表示と～ 58
IPレジスタ 42,47
80386 528
サブルーチンと～ 186
IRETD命令
80386 548
IRP命令
繰り返しブロックと～ 453
IRPC命令
繰り返しブロックと～ 453

【J】

/Jオプション 74
JA命令 181
JAE命令 181
JB命令 181
JBE命令 181
JC命令 181
JCXZ命令 183,272
80386 547
JECXZ命令
80386 547
JE命令 180,181
JEMULオプション 71
JG命令 181
JGE命令 181
JL命令 181
JLE命令 181
JMP命令 176
ラベルと～ 108
JNA命令 181
JNAE命令 181
JNB命令 181
JNBE命令 181
JNC命令 181
JNE命令 181
JNG命令 181
JNGE命令 181
JNL命令 181
JNLE命令 181
JNO命令 181
JNP命令 181
JNS命令 181
JNZ命令 181
80386 547
JO命令 181
JP命令 181
JPE命令 181
JS命令 181
JUMPS疑似命令 441
JZ命令 181,184

【K】

/KHオプション 75
/KSオプション 76

【L】

/Lオプション 69,77,81
$Lコンパイラ指令
(Turbo Pascal) 363
/LAオプション 77
LABEL疑似命令 147,438

LARGE演算子
80386 と～ 519
LEA命令とMOV OFFSET 291
LEAVE命令(80186) 508
LES命令 543
LFCOND疑似命令 88
%LINUM疑似命令 245
%LIST疑似命令 241
LOCAL疑似命令
Turbo Cと～ 341
マクロ内の～ 462
LOCALS疑似命令 439
LODS命令 208
複数セグメントと～ 503
LODSB命令 208
セグメントオーバライドと～ 434
LODSW命令 209
long整数 134
LOOP命令 36,183
80386 546
キャリフラグへの影響 281
問題点 271-272
LOOPD命令(80386) 546
LOOPDE命令(80386) 546
LOOPDNE命令(80386) 546
LOOPE命令 183
80386 546
LOOPNE命令 183
80386 546
LOOPZ命令 184
LOW演算子
Idealモード 571
.LSTファイル 65,230

【M】

MACRO疑似命令 462
%MACS疑似命令 243
MAKE.EXE 6
MASK演算子 477
MASM疑似命令 562
MASMとの互換性 1,97,241,246,288
Idealモードとの比較 533-589
OFFSET演算子のバグ 491
Turbo Cと～ 302
イコール定義 566
構造体 465
式 567
セグメントグループ 491,577
セグメントの順序付け 124
リスト制御疑似命令 574
ローカルラベル 439
MASMモード → MASMとの互換性
MEMORY結合型 485
/MLオプション 78,228
.MODEL疑似命令 124,497
MOV命令 151-159
80386 544
LEA命令との比較 291
アドレシングモードと～ 115
ストリング命令との比較 208
前方参照 448
MOVS命令 211
ESレジスタと～ 432
MOVSB命令 212
80386 548
MOVSD命令
80386 547
MOVSW命令 212

MOVSX命令 538
MOVZX命令 538
MS-DOS → DOS
/MUオプション 79
MUL命令 166
問題点 280
/MXオプション 80

【N】

/Nオプション 81,235
NEAR型 147
NEAR PTR演算子と～ 154
nearサブルーチン 188
nearデータ 124
Idealモード 579
NEAR PTR演算子 154
NEG命令 169
%NEWPAGE疑似命令 245
%NOCONDS疑似命令 242
%NOCREF疑似命令 241
%NOCTLS疑似命令 230
NOEMUL疑似命令 71
%NOINCL疑似命令 242
NOJUMPS疑似命令 444
%NOLIST疑似命令 241
NOLOCALS疑似命令 439
%NOMACS疑似命令 243
%NOSYMS疑似命令 244
NOT命令 169
NOTHINGキーワード 130
NOTHING疑似命令 491
%NOTRUNC疑似命令 245
%NOUREF疑似命令 244
NULデバイス 66

【O】

.OBJファイル 1,11
OBJXREF.COM 6
OFFSET演算子
MASMモードとIdealモード 577
問題点 284,291
OR命令 169
OS/2 515
%OUT疑似命令 247
OUT命令 37,160
OUTS命令(80186) 510

【P】

/Pオプション 82
%PAGESIZE疑似命令 245
Pascal → Turbo Pascal
PC-DOS → DOS
%PCNT疑似命令 245
POP命令 40,159
POPA命令(80186) 507
POPAD命令
80386 549
POPFD命令
80386 549
%POPLCTL疑似命令 245
PRIVATE結合型 485
PROC疑似命令 149,438
高級言語と～ 125
サブルーチンと～ 188,260
Prolog → Turbo Prolog
PSP 358
PTR演算子
Idealモード 572
PUBLIC疑似命令 222

Idealモード 580
Turbo Pascalと～ 364
PUBLIC結合型 485
PUSH命令 40,159
80186 512
PUSHA命令(80186) 507
PUSHAD命令
80386 549
PUSHFD命令
80386 549
%PUSHLCTL疑似命令 245
PWORD型 147

【Q】

QWORD型 147

【R】

/Rオプション 71,83
.RADIX疑似命令 140
RCL命令 173
RCR命令 173
README.COM 6
RECORD疑似命令 472
REPプレフィクス
セグメントオーバライドと～ 436
文字列のオーバラン 268
問題点 273
REPE命令 215
REPNE命令 215
REPNZ命令 215
REPT疑似命令 450
DUP疑似命令との比較 451
REPZ命令 215
RET命令
PROC疑似命令と～ 260
サブルーチンと～ 187,259
ROL命令 173
ROR命令 173

【S】

/Sオプション 68,84,488
-sオプション(Turbo C) 301
SAL命令 172
SAR命令 173
SBB命令 163
SCAS命令 214
ESレジスタと～ 432
繰り返し 273
SCASB命令 214
SEG演算子
Turbo Pascalと～ 368
SEGMENT疑似命令 128,483
.SEQ疑似命令 488
SEQ疑似命令 68
SET命令 542
SETNC命令 542
SETS命令 542
SFCOND疑似命令 88
SHL命令 172
80386 513
SHLD命令 540
SHORT演算子 177
前方ジャンプと～ 447
short整数 134
SHR命令 173
80186 513
レコードと～ 475
SHRD命令 541
SIレジスタ 37,47
ストリング命令 212

メモリポインタ 113
SIZE演算子
Idealモード 573,580
SMALL演算子
80386 と～ 518
SPレジスタ 40,48
SSレジスタ
LESレジスタと～ 543
～へのメモリポインタ 39
メモリオペランドと～ 432
.STACK疑似命令 120,263
STACK結合型 485
STD命令 272
STOS命令 210
ESレジスタと～ 432
複数セグメントと～ 503
STOSB命令 39,210
STOSW命令 210
～の落とし穴 278
STRUC疑似命令 465
SUB命令 162
%SUBTTL疑似命令 244
%SYMS疑似命令 244
.SYMTYPE演算子
Idealモード 571

【T】

/Tオプション 85
TASM.CFGファイル 92
TASM.EXE 6
TBYTE型 147
TCREF 1
TCREF.EXE 6
TEST
ANDとの比較 477
%TEXT疑似命令 245
TFCOND疑似命令 88
%TITLE疑似命令 244
Idealモード 576
TLIB.EXE 6
TLINK 1,11,335,350
Turbo Cのバージョン 296
セグメントの順序付け 487
TLINK.EXE 6
TOUCH.COM 6
%TRUNC疑似命令 245
Turbo Basic 393-412
CALLルーチン 398
CALL ABSOLUTE 405
CALL INTERRUPT 409
インラインアセンブラ 400
スタック 399
引数の受け渡し 395
文字列 404
Turbo C 293-354
-1オプション 304
-bオプション 301
-sオプション 301
#pragma指令 301
80186と～ 507
80186/80286プロセッサ 304
ARG疑似命令と～ 341
LOCAL疑似命令と～ 339
MASMと～ 302
Pascalのコール規定 348
アセンブリ言語と～の比較 24
インラインアセンブラ 294-319
コメント 306
制限 312
セミコロン 305

フォーマット 305
大文字小文字の区別 79,331
オペランドの制限 312
外部シンボル 333
コードセグメント 322
構造体 307,344
ジャンプ 312
セグメント疑似命令と～ 323
データ型 332
～とのリンク 289,350
～のアセンブラモジュール 71
パスの命名規定 229
パブリックファンクションと～ 330
引数の受け渡し 335
ピリオド演算子 468
浮動小数点エミュレーション 71
メモリモデル 322
戻り値 343
レジスタの退避 315,343
Turbo Debugger 1,89
Turbo Link → TLINK
Turbo Pascal 355-392
EXTRN疑似命令 364
near/farコール 362
PUBLIC疑似命令 364
最適化 368
スタック 359
セグメントの付加情報 368
データセグメント 358
ヒープ 359
引数の受け渡し 370
ファンクションの実行結果 378
メモリマップ 356
メモリモデル 356
戻り値 378
レジスタの使いかた 361
ローカルデータの定義 379
Turbo Prolog 413-430
グローバル述語 414
データ型 415
ファンクタ 427
フローパターン 416
複数～ 423
リスト 427
.TYPE演算子
Idealモード 571

【U】

UNION疑似命令 480
UNKNOWN型 147
%UREF疑似命令 244
USE16型セグメント 518,550
USE32型セグメント 519,550

【V】

/Vオプション 67,85

【W】

/Wオプション 86
WIDTH演算子 476
WORD PTR演算子 153
WORD型 147

【X】

/Xオプション 88
XCHG命令 159
.XCREF疑似命令 241
XLAT命令
～のオペランド 278
XOR命令 169

.XRFファイル 66

【Z】

/Zオプション 88
/ZDオプション 89
/ZIオプション 90

【あ】

アーキテクチャ(コンピュータ) 21
アキュムレータ 34,208
アセンブラ命令のフォーマット 96
アセンブリ言語
Turbo Cとアセンブリ言語 24
定義 23
アセンブル
インライン～
Turbo Basic 400
Turbo C 294-319
最初のプログラムを～する 9
条件～ 248
アドレシングモード 109
80386 531
アンダライン
Turbo Cと～ 330
アンパサンド
パラメータと～ 458
イコール定義 200
インクリメント 164
インクルードファイル 228
GLOBAL疑似命令と～ 226
パスの設定 73
リストの不表示 242
インラインアセンブラ
Turbo Basic 400
Turbo C 294-319
制限 312
フォーマット 305
エクストラセグメント 48
セグメントオーバライドと～ 434
エラーメッセージ 15
条件～ 255

ソースファイル行の表示 88
円記号(¥) 229
インクルードファイルと～ 229
演算子 → 各演算子の説明を参照
Idealモードと
MASMモードの比較 584
式と～ 106
マクロ 463
オーバフローフラグ 33
条件ジャンプと～ 180
大文字小文字の区別
Turbo C 331
アセンブラルーチンと～ 80
パブリックラベルと～ 223
落とし穴 258-291
CXレジスタ 271
LOOP命令 271
OFFSET演算子 284
REPプレフィクス 273
REP文字列のオーバラン 268
Turbo Cモジュールとのリンク 289
オペランドの順番が逆 262
キャリフラグ 280
サブルーチン 259,261,264
条件ジャンプ 266
乗算 279
スタック領域の確保 263
ストリング命令 267,280
ストリング命令のオペランド 277
ストリング命令の
デフォルトセグメント 273
セグメントグループ 287
セグメントレジスタの一巡 284
ディレクションフラグ 272
バイトからワードへの変換 275
複数プレフィクス 277
フラグ 272
プログラムの終了 258
メモリ変数 282
文字列の比較 273
リターン 260
レジスタの破壊 264
割り込みハンドラ 286
オブジェクトファイル
セグメントの順序付け 68,84
～内の行番号情報 89
～内のデバッグ情報 90
オブジェクトモジュール
定義 10
オプション
→ コマンドラインのオプション
オフセット
$記号と～ 205
オペランド
Idealモード 567
Turbo Cにおける制限 312
～のあるストリング命令 218
～の順序 262
ソース～ 103
定義 102
定数～ 104
デスティネーション～ 103
メモリ～ 109
BPレジスタと～ 432
ESレジスタと～ 432
文字～ 105
ラベル 107
レジスタ～ 103

【か】

開始アドレス
　END疑似命令と～ 101
解析順序
　Idealモード 575
外部シンボル 80
カウンタ
　CXレジスタと～ 35
拡張情報の表示 67
加算 162
型 → データ型
型指定子 147
型チェック
　Idealモード 561
画面モード
　ANSI.SYS 61
　BIOSと～ 60
カラー
　BIOSと～ 60
カラーグラフィックスアダプタ → CGA
間接アドレシング 112
間接コマンドファイル 91
感嘆符(!)演算子
　マクロ内の～ 463
簡略セグメント疑似命令 120-127
　80386 522
　ASSUMEと～ 493,495
キーの読み込み
　DOSファンクション 57
キーワード 78
機械語 23
疑似命令
　Idealモード 574
　条件アセンブル～ 249
　条件エラー～ 255
　シンボル 74
　スタートアップ 74
　セグメント～ 94,120,127,483-503
　データ定義～ 142
　定義 100
　～のピリオド 574
　プロセッサ型指定～ 506
　プロセッサ制御～ 574
　文字列スペース 76
基数の指定
　.RADIX疑似命令と～ 140
既定義シンボル → シンボル
基本入出力システム → BIOS
疑問符(?)
　初期値不設定記号 146
　ラベル内の～ 97
キャリフラグ 33
　32ビット算術演算 163
　予期せぬ結果 280
境界チェック 509
共用体
　Idealモードと
　　MASMモードの比較 566
クオッドワード 135
　ダブルワードへの変換 539
グラフィクス 61
繰り返し命令 212,450
グローバル述語(Turbo Prolog) 414
クロスリファレンス
　～リストの作成 65
　リストファイル中の～ 69,238
クロスリファレンステーブル 237
警告メッセージ 15
　～の発生 85

マイルドな～ 86
結合型 237
AT 484
COMMON 485
MEMORY 485
PRIVATE 485
PUBLIC 485
STACK 485
言語要素 96
減算 162
コードセグメント 47
Turbo C 322
Turbo Pascal 358
複数の～ 500
コードチェック 82
コール
前方参照と～ 449
高級言語
80186 と～ 508
～とアセンブラモジュールのリンク
124-128,132,289,293-430
セグメントグループ 490
セグメントの順序付け 487
戻り値 191
構造体 465-482
DUP演算子と～ 468
Idealモード 472,566,580
MASMモード 465
Turbo C 307,344
初期化 471
前方参照 465
ピリオド演算子 468
構文
コマンドラインの～ 64
コマンドファイル
間接～ 91
コマンドラインのオプション 67～90
コマンドラインの構文 64
ヘルプ画面 72
コメント(注釈) 116
Idealモード 582
インラインアセンブラの～ 306
コロン
ラベル内の～ 97
コンパイラオプション
→ 各オプションの説明を参照
コンパクトメモリモデル 125
コンフィギュレーションファイル 92

【さ】

再帰 186
最適化(Turbo Pascal) 368
サインフラグ
条件ジャンプと～ 180
サブルーチン 186
far 188
near 188
RET命令 259
マクロとの比較 456
レジスタの退避 264
ローカルラベル 437
算術演算 21,162
符号付き数値 154
式 106
Idealモード 567
～内の演算子 106
変数の初期化 145
時刻 61

システムソフトウェア
　IBM PC 56
システムタイマ 61
実行
　END疑似命令と～ 101
実数 139
実モード(80286) 516
シフト 172
　複数ワードにまたがる～(80386) 540
ジャンプ
　80386 528,547
　FAR PTRと～ 178
　short～ 177
　Turbo Cの制限 312
　条件～ 179
　　サイズ 441
　　問題点 266
　　ローカルラベルと～ 439
　前方参照 446
　無条件～ 176
終了 59
　DOSファンクションの～ 59
　END疑似命令と～ 95,101
　問題点 258
述語
　Turbo Prolog 425
　アセンブリ言語 425
条件アセンブル
　～疑似命令 249
　　マクロ命令と～ 460
条件エラー疑似命令 255
条件疑似命令
　Turbo Cの～ 250
　アセンブラと他の言語の～ 266
　リスト内の～ 87
　　～の不表示 242
条件ジャンプ → ジャンプ
条件テスト(80386) 542
乗算 166
　80386 549
　AXレジスタと～ 34
　DXレジスタと～ 37
　REPT疑似命令と～ 452
　SHL命令と～ 172
　～の落とし穴 279
　符号付き～ 167
　符号なし～ 166
初期化
　構造体 471
　式と～ 145
　データの～ 142
　配列の～ 143
　文字列の～ 144
　ラベル 145
　レコード変数 474
初期値を設定しないデータ定義 146
除算 166
　AXレジスタと～ 34
　DXレジスタと～ 37
　REPT疑似命令と～ 452
　符号付き～ 167
　　SAR命令 173
　符号なし～ 167
　　SHR命令 173
シリアル通信 61
シンボル
　@CodeSize 126
　@curseg 126
　@data 121
　@fardata 126

@fardata? 126
@FileName 126
Idealモード 564
大文字 80
大文字小文字の区別 79,80
外部～ 80
Idealモード 580
Turbo Cと～ 333
制限 69
定義 69
パブリック～ 80
未参照～ 244
ローカル～
Idealモード 583
シンボルテーブル
～の不表示 81,235,244
リストファイル 235
クロスリファレンス 69
数値
インクルードファイルと～ 229
符号付き～ 154
符号なし～ 154
ラベル内の～ 96,200
数値演算コプロセッサ 71,557
スタック
80186 命令 507
MOV命令と～ 159
Turbo Basic 399
Turbo Pascal 359
結合型と～ 485
～セグメント 48
セグメントオーバライドと～ 436
～のサイズ 120
～の定義 263
引数の受け渡し 190
～ポインタ 40,48
スタックセグメントレジスタ(SS)
～へのメモリポインタ 40
ストリング命令 208
80386 547
BPレジスタ 39
CMPS 217
DIレジスタ 38
ESレジスタ 124
LODS 208
MOVS 211,271
REPプレフィクス 212,215
SCAS 214
SIレジスタ 37
STOS 210
インクリメント 272
エクストラセグメント 48,435
繰り返し～ 212,215,268
～と非ストリング命令の混在 436
～の落とし穴 280
～のオペランド 277
ディレクションフラグ 259
データ転送～ 208
デクリメント 272
バイトとワードの比較 218
複数プレフィクス 277
フラグ 215
レジスタへの影響 280
スピーカ 61
スモールメモリモデル 127
スラッシュ
インクルードファイルと～ 229
整数
10バイト 135
16ビット 134

32ビット 134
long 134
short 134
セグメント
80386 518,550
Turbo Cと～ 322
USE16 518
USE32 518
～アラインメント
80386 522
～型 237,483
アルファベット～オーダ 488
エクストラ～ 48
～オーバライド
～プレフィクス 432
カレント～ 128
～疑似命令 95,120,483-503
80386 522
Turbo Cと～ 323
簡略～ 120-127,493,495
シンボルと～ 128
高級言語と～ 132
標準～ 127-131
グループ 489
Idealモードと～ 560,577
MASMモード 577
Turbo Cと～ 287
問題点 287
～結合型 237,484
コード～ 47,120,500
Turbo Pascal 358
シーケンシャル～オーダ 68,84,488
スタック～ 48
データ～ 47,120,126,500
～の終わり 128
～の先頭 130
～の順序付け 63,124,486,487
～のネスティング 487
付加情報
Idealモードと MASMモード 560
複数～ 500
複数～へのアクセス 436
～名 45,486
リスト表示 235
～レジスタ 43
→ 各レジスタの説明を参照
80386 529
CS 47
DS 47,121
ES 48,123
FS 529
GS 529
SS 48
～間でのデータ転送 157
～レジスタの一巡 284
～へのメモリポインタ 38,39
セグメント：オフセットポインタ 134,500
セグメントのグループ化 287,489
Idealモード 577
絶対値 224
セミコロン(；) 64
インラインアセンブラと～ 305
セミコロン(；)演算子
マクロ内の～ 463
ゼロ除算のエラー 168
ゼロフラグ 33
条件ジャンプと～ 180
ループと～ 184
前方参照 446
構造体 465

マクロへの～ 463
ソースファイル
インクルードファイル 73
シンボル 69

【た】

大カッコ([]) 112
Idealモード 567
MASMモード 567
ダイレクトアドレシングと～ 111
大カッコ演算子([])
マクロ内の～ 463
タイニィメモリモデル 125
タイマ 61
ダイレクトアドレシング 111
他のアセンブラとの互換性
226,241,254,288
→ MASMとの互換性も参照
ダブルワード 134
クオッドワードへの変換 539
ワードへの変換 157
単価記号(@) 91
チュートリアル 7-61
通信ソフトウェア 61
データ
～型 133,147
Turbo C 332
Turbo Prolog 415
UNKNOWN 147
複数の～ 480
ラベル 224
～サイズ 152
～サイズの変換 155
BYTE PTR演算子 153
WORD PTR演算子 153
ストリング命令 218
初期化されない～ 146
～スキャン 214,537
前方参照 448
～の交換(スワップ) 159
～のコピー 151
～の初期化 142
～の定義 133-161
～の転送 151-161
80386 538
ストリング命令 208
データ構造 → 構造体
データサイズ → データ
データセグメント 47,133-161
.CONST疑似命令 126
.DATA?疑似命令 126
.DATA疑似命令 120
.FARDATA疑似命令 126
Turbo Pascal 358
複数の～ 500
データ転送 151-161
データのコピー(複写) 151
定数
$ 205
～オペランド 104
～式 106
～の使用制限 105
文字～ 140
ディスクオペレーティングシステム
→ DOS
ディレクションフラグ 33
ストリング命令と～ 208
設定エラー 272
テキスト → 文字列
デクリメント 164

デバッグ 89
トークン
Idealモード 564
等号(=) 69
トラップフラグ 33
ドル記号($) 205
ラベル内の～ 97

【な】

名前 96
ニーモニック 100
入出力 → I/O

【は】

パーセント記号(%)
Idealモード 574
パーセント演算子(%)
マクロ内の～ 463
ハードウェアへのダイレクトアクセス 61
バイト 133
BYTE PTR演算子 153
CMPSB命令 217
LODSB命令 196
MOVSB命令 199
SCASB命令 214
STOSB命令 210
条件設定 542
ストリング命令 218
ワードへの変換 155,275
配列
～の初期化 143
バッファ
～のコピー 210
～のサイズ 67
パブリックファンクション
Turbo Cと～ 330
パブリックラベル 222-227
Idealモード 580
パラメータの受け渡し
Turbo Basic 395
Turbo C 335
Turbo Pascal 370
スタック 190
テスト 252
レジスタ 190
パリティフラグ
条件ジャンプと～ 180
汎用レジスタ 33
→ 各汎用レジスタの項を参照
80386 527
AX 34
BP 39
BX 35
CX 35
DI 38
DX 37
SI 37
SP 40
符号反転 169
ヒープ(Turbo Pascal) 359
比較命令 214
～の繰り返し 273
引数(パラメータ)
仮引数 456
マクロと～ 456
日付 61
ビット 133
～の扱い 166
～のシフト 172

～のスキャン(80386) 537
～のテスト(80386) 536
ビットフィールド 472
ヒュージメモリモデル 125
標準セグメント疑似命令 127-132
ピリオド
～演算子 468
疑似命令内の～ 574
構造体内の～
Idealモード 570
ラベル内の～ 96
Idealモード 564
ブール代数 169
ファイル
.ASM～ 65
インクルード～ 228
間接～ 91
コンフィギュレーション～ 92
リスト～ 65,230
→ リストファイル
ファンクタ(Turbo Prolog) 427
複数のプレフィクス 277
符号付き乗算 167
符号付き除算 169
SARと～ 173
符号付き数値 154
符号付きデータ変換命令 538
符号なし乗算 166
符号なし除算 167
SHRと～ 173
符号なし数値 154
符号の変更 169
不等号カッコ(< >) 204
EQU疑似命令と～ 204
条件疑似命令と～ 254
マクロ内の～演算子 463
浮動小数点
IdealモードとMASMモード 566
～エミュレータ 71
～演算子 61
～数 134,139
～命令 2
フラグレジスタ 31
80286 516
80386 528
ストリング命令 215
問題点 281
プラス記号(+) 65
プラス演算子(+) 468
プリンタへの出力
最初のプログラムの～ 16
プレフィクス 277
セグメントオーバライド～ 432
複数～ 277
ローカルラベル 440
フローパタン(Turbo Prolog) 417
複数の～ 423
プログラミング上のエラー 258-291
プログラミングの落とし穴 258
プログラムセグメントプレフィクス
(PSP) 358
プログラムの構造 93-197
プログラムの終了 59,95,101
問題点 258
プロセッサ
定義 22
プロテクトモード → 保護モード
分岐命令 176
命令ポインタと～ 42
ヘルプ画面の表示 72

変換
データサイズの～ 155
バイトからワードへの～ 155
問題点 276
ワードからダブルワードへの～ 157
変数
インクリメント/デクリメント 164
インラインアセンブラと～ 314
初期化されない～ 146
符号の変更 169
メモリ 142
文字列への変換 58
レコード 473
ポインタ
32ビットポインタ
Turbo Prolog 415
I/O
DXレジスタ 160
DXレジスタと～ 37
構造体への～
Idealモード 571
セグメント：オフセット 134
セグメントと～ 45
定義 23
メモリ～ 35,112
BPレジスタと～ 39
CSレジスタと～ 47
DIレジスタと～ 38
DSレジスタと～ 47
ESレジスタと～ 47
SIレジスタと～ 37
SPレジスタと～ 40
SSレジスタと～ 48
保護モード 82
80286 516
80287 557
保護モードの命令
80286 516

【ま】

マイクロソフトアセンブラ
→ MASMとの互換性
マクロ命令 454
コメント
Idealモード 582
サブルーチンとの比較 456
条件アセンブル疑似命令 460
条件エラー疑似命令 255
前方参照 463
～内の演算子 463
～内のラベル 462
ネストしたセグメント 486
～の展開
EXITM疑似命令 461
リストの不表示 243
マニュアルの表記法 4
ミディアムメモリモデル 125
無条件ジャンプ → ジャンプ
命令セット → 各命令の説明を参照
8086 49
命令ニーモニック → ニーモニック
命令プレフィクス 212
命令ポインタ 42,47
80386 528
メッセージ
アセンブル中の～ 234
～の不表示 85
メモリ
～アドレシング
80386 532

不等号カッコと～ 116
～モード 109
ラージブロック 46
～アドレスの命名 147
オペランド 109
BPレジスタと～ 432
DSレジスタと～ 432
ESレジスタと～ 432
間接アドレシング 113
～管理(80286) 516
ストリング命令と～ 38,39
セグメント名 45
ダイレクトアドレシング 111
～の確保 147
～のスキャン 214
～のセグメント 43
～ブロック 143,213
～ブロックのパディング 213
ブロックの比較 214
～変数
～変数の定義 142
問題点 282
ポインタ 112
BPレジスタと～ 39
BXレジスタと～ 35
CSレジスタと～ 47
DIレジスタと～ 38
DSレジスタと～ 47
ESレジスタと～ 48
SIレジスタと～ 37
SPレジスタと～ 40
SSレジスタと～ 48
～マッピング 31
～モデル
FAR型と～ 148
.MODEL疑似命令と～ 124
NEAR型と～ 148
PROC型と～ 149
Turbo C 322
Turbo Pascal 356
Turbo Prolog 415
セグメントと～ 495
文字
DOSファンクション 58
～オペランド 105
～の表示 58
文字列の初期化 144
文字定数 140
文字の表示 → 文字
モジュール化プログラミング 186,220
END疑似命令と～ 102
EXTRN疑似命令と～ 223
.FARDATAセグメントと～ 498
GLOBAL疑似命令 226
PUBLIC疑似命令 222
ローカルラベル 437
文字列
Turbo Basic 404
引用符(Idealモード) 576
～の初期化 144
～の比較 273
～の表示 247
ラベルへの～割り当て 200
モデルが混在するプログラミング
セグメント疑似命令と～ 132

【ら】

ラージメモリモデル 125
ラベル
=疑似命令と～ 206

EQU疑似命令と～ 200
EXTRN疑似命令と～ 223
Idealモード 439
MASMモード 439
PUBLIC疑似命令と～ 222
演算子としての～ 107
外部～ 224
Turbo Cの～ 330
条件ジャンプと～ 439
数値/文字列のイコール定義 200
データ型 224
二重定義 251
～の再定義 206
～の条件 96
～のリスト表示 235
変数の初期化 145
マクロ内の～ 462
未定義～ 256
メモリアドレスの～ 147
モジュールと～ 222-227,437
ローカル～ 437
リスト(Turbo Prolog) 427
リストファイル 66,230-246
%CREF疑似命令 240
%NOCREF疑似命令 240
行の不表示 241
クロスリファレンス情報 69,237
シンボル テーブル
～の不表示 81,235
タイトル 244
Idealモード 576
～内の高級言語命令 77
～内の偽条件 88
～内の数字 229
～の作成 77
～のページサイズ 245
リスト形式 244
リスト制御疑似命令 241
Idealモード 573
MASMモード 574
リンク → TLINK
Turbo C 289,335,350
高級言語 132,293-412
戻り値 191
最初のプログラム 11
ループ処理 182
CXレジスタと～ 35
レコード 475
レジスタ → 各レジスタの説明を参照
32ビット～ 528
80386 526
8086 31
Turbo Pascal 361
インクリメント/デクリメント 164
～オペランド 103
～のクリア(0 セット) 172
～の退避 192,264
～の退避(Turbo C) 315,343
パラメータの受け渡し 190
ローカルシンボル
IdealモードとMASMモード 583
ローテーション 172
論理演算 21,169

【わ】

ワード 134
CMPSWと～ 217
LODSWと～ 209
MOVSWと～ 212
SCASWと～ 214

STOSWと～ 210
WORD PTR演算子 153
ストリング命令 218
ダブルワードへの変換 157
バイトへの変換 156
ワイルドカード → DOS
割り込み
～0 168
ゼロ除算の～ 168
割り込みハンドラ
レジスタの退避 286
割り込みフラグ 33

1 TAUTAIL 890318

Turbo Assembler 1.0 ユーザーズガイド

1989年 4月 1日　初版発行

著者　**Borland International**

編訳・発行　**(株)ボーランド ジャパン**

〒107 東京都港区南青山7-8-1 小田急南青山ビル

電話 (03)486-1400(代表)

製品に関するお問い合わせは,

下記代理店までお願い致します。

(株)マイクロソフトウェア アソシエイツ

〒107 東京都港区南青山7-8-1 小田急南青山ビル

電話 (03)486-1411(代表)

(03)486-1403(サポートセンター)

FAX (03)486-8905

Printed in Japan

落丁,乱丁はお取り替えいたします。

TURBO ASSEMBLER®
BORLAND
Copyright © 1988 Borland International, Inc. All rights reserved. Licensed material—program property of Borland International, Inc. 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95066-0001. A *Turbo Language*™ product. Unauthorized use, duplication, or distribution is strictly prohibited by federal law.
PART# 15LB-ASD01 BOR 0855
DISK 1
L1E0027049
TURBO ASSEMBLER & DEBUGGER®
BORLAND

USER'S GUIDE

TURBO ASSEMBLER®

BORLAND